TotalView™

Multiprocess Debugger

User’s Guide

Version 3.8.1

December, 1998

Dolphin Interconnect has prepared this manual for the exclusive use of its customers, personnel, and licensees. The information in this manual is subject to change without notice, and should not be construed as a commitment by Dolphin Interconnect. Dolphin Interconnect assumes no responsibility for any errors that appear in this document.

Revision 3.8.1: Printed December 1998

About This Guide

This guide describes how to use TotalView, a source-level and machine-level debugger with an easy-to-use interface (based on the X Window System) and support for debugging multiprocess programs. The guide assumes that you are familiar with the C programming language, UNIX operating systems, the X Window System, and the processor architecture of the platform on which you're running TotalView.

This guide covers the general use of TotalView on any platform. Most of the examples and illustrations in this guide show TotalView running on a workstation. To learn about the specifics of running TotalView on your platform, refer to Appendix A, “Compilers and Environments,” on page 303, Appendix B, “Operating Systems,” on page 321, and Appendix C, “Architectures,” on page 333.

Getting Started

To get started quickly with TotalView:

- Install the software.

  The *TotalView Installation Guide* provides instructions.

- Learn the basics of TotalView.

  Chapter 2, “TotalView Basics,” on page 15 and Chapter 6, “Debugging Programs,” on page 115 provide instructions.

Supported Platforms

TotalView is available for a variety of platforms and can be used to debug programs on the native platform or on remote systems, such as parallel processors, supercomputers, or digital signal processor boards.

If TotalView is not yet available for your system configuration, please contact Dolphin Interconnect about porting TotalView to suit your needs:

ToolWorks Group
Dolphin Interconnect Solutions, Inc.
111 Speen Street
Framingham, MA 01701-2090
Internet E-mail: **toolworks@dolphinics.com**
1-800-856-3766 in the United States
(+1) 508-875-3030 worldwide

Reporting Problems

Please contact us if you have problems installing TotalView, questions that are not answered in the product documentation or on our Web site, or suggestions for new features or improvements.

Our Internet E-Mail address is: **tv-support@dolphinics.com**
1-800-856-3766 in the United States
(+1) 508-875-3030 worldwide

If you are reporting a problem, please include the following information:

- The **version** of TotalView
- The **platform** on which you’re running TotalView
- An **example** that illustrates the problem
- A **record** of the sequence of events that led to the problem

See the TotalView Release Notes for complete instructions on how to report problems.

Typographical Conventions

This guide uses the following conventions to present information:

bold An exact filename, command, or user input.

italic In examples indicates a variable or a value that you supply. In text, emphasizes important words or phrases.

`typewriter` Computer output.

Control-Z Press the keys simultaneously; for example, hold down the Control key and press the Z key.

^Z Shorthand for Control-Z.

Esc Z Press the first key and then the second; for example, press the Escape key and then press the Z key.

M-I Shorthand for Meta-I. (The Meta key varies with your platform; usually it is the Alt key.)

[ ] Optional items in command syntax descriptions.

. . . Repetition of the previous command or input.

(G) The keyboard equivalent for a command in parentheses; for example, **Go Group (G)**.

Contents

CHAPTER 4:

CHAPTER 5:

CHAPTER 6:

APPENDIX C:

List of Figures

List of Tables

CHAPTER 1: Introduction

The TotalView debugger is part of a suite of software development tools for debugging, analyzing, and tuning the performance of programs, including multiprocess multithreaded programs. In addition to TotalView, you can purchase the TimeScan Performance Analyzer to generate and analyze event logs. This chapter highlights the features of TotalView and includes the following sections:

- TotalView's advantages
- TotalView's windows
- Examining source and machine code
- Controlling processes and threads
- Using Action Points
- Examining and manipulating data
- Visualizing array data
- Distributed debugging
- Debugging multiprocess and multithreaded programs
- Context-sensitive help

TotalView's Advantages

TotalView provides many advantages over conventional UNIX debuggers (such as **dbx**, **gdb**, and **adb**):

- You can learn TotalView quickly and be more productive because of its graphical interface (based on the X Window System). TotalView's interface provides windows, pop-up menus, and a context-sensitive help system. You can enter most commands with the mouse. Further, with TotalView's interface, you can already see a lot of useful information without entering any commands.
- You can debug *multiprocess multithreaded* programs because TotalView can manage multiple processes, and multiple threads within a process. TotalView displays each process in its own window, showing the source code, stack trace, and stack frame for one or more threads in the process. You can display all process windows simultaneously and perform all debugging tasks across processes.
- You can debug *remote* programs over the network because of TotalView's distributed architecture, as shown in Figure 1. Remote programs are programs that run on a different machine from TotalView, while native programs are programs that run on the same machine as TotalView.

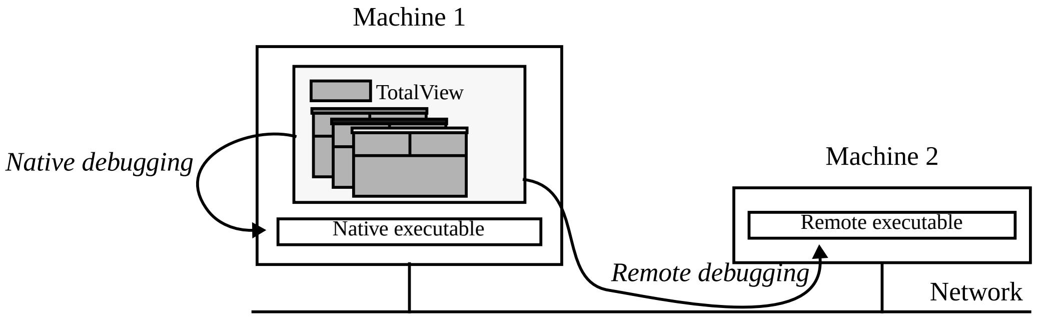


Figure 1. Debugging a Remote Program with TotalView

- You can debug *distributed* programs over the network because TotalView can manage multiple remote programs and multiprocess multithreaded programs simultaneously, as shown in Figure 2. Distributed programs are programs that run on a group of separate homogeneous machines.

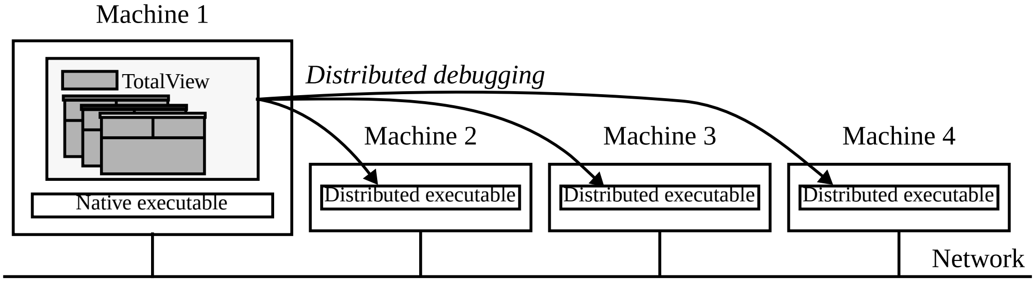
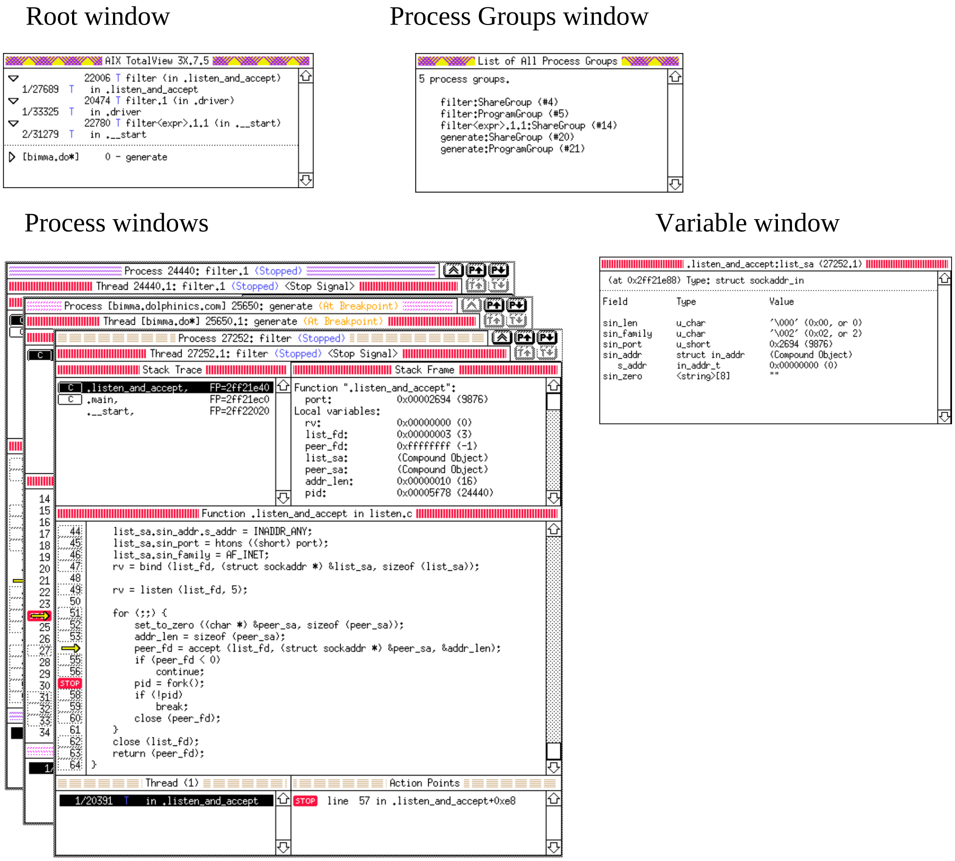


Figure 2. Debugging a Distributed Program with TotalView

- You can acquire processes automatically for several popular runtime libraries, such as HPF, PVM, PE, and MPI. Parallel and distributed programs run in many processes, and your debugger must know about them for you to debug them correctly. When you start TotalView in HPF, MPI, PE, or PVM, TotalView automatically detects these processes and attaches to them so you don't have to attach to them manually. This process is called *automatic process acquisition*. If the process is on a remote machine, automatic process acquisition also automatically starts the TotalView debugger server.
- You can write source code fragments within TotalView and insert them temporarily into the program you're debugging. On some platforms, you can write machine code fragments as well. This feature can save you time in testing bug fixes.
- You can debug code that was not compiled with the **–g** switch or for which you don't have access to the source file because TotalView provides machine-level debugging features.
- You can attach to running processes, so you can debug processes that were not started under TotalView.

TotalView’s Windows

TotalView displays extensive information in its windows, as shown in Figure 3.



Root window

```
AIX TotalView 3X.7.5
▽               22006 T filter (in .listen_and_accept)
   1/27689  T    in .listen_and_accept
▽               20474 T filter.1 (in .driver)
   1/33325  T    in .driver
▽               22780 T filter<expr>.1.1 (in .__start)
   2/31279  T    in .__start
▷ [bimma.do*]     0 - generate
```

Process Groups window

```
List of All Process Groups
5 process groups.

    filter:ShareGroup (#4)
    filter:ProgramGroup (#5)
    filter<expr>.1.1:ShareGroup (#14)
    generate:ShareGroup (#20)
    generate:ProgramGroup (#21)
```

Process windows

```
Process 24440: filter.1 (Stopped)
Thread 24440.1: filter.1 (Stopped) <Stop Signal>
Process [bimma.dolphinics.com] 25650: generate (At Breakpoint)
Thread [bimma.do*] 25650.1: generate (At Breakpoint)
Process 27252: filter (Stopped)
Thread 27252.1: filter (Stopped) <Stop Signal>
```

```
Stack Trace
C  .listen_and_accept,     FP=2ff21e40
C  .main,                  FP=2ff21ec0
   .__start,               FP=2ff22020
```

```
Stack Frame
Function ".listen_and_accept":
  port:              0x00002694 (9876)
Local variables:
  rv:                0x00000000 (0)
  list_fd:           0x00000003 (3)
  peer_fd:           0xffffffff (-1)
  list_sa:           (Compound Object)
  peer_sa:           (Compound Object)
  addr_len:          0x00000010 (16)
  pid:               0x00005f78 (24440)
```

```
Function .listen_and_accept in listen.c
        list_sa.sin_addr.s_addr = INADDR_ANY;
        list_sa.sin_port = htons ((short) port);
        list_sa.sin_family = AF_INET;
        rv = bind (list_fd, (struct sockaddr *) &list_sa, sizeof (list_sa));

        rv = listen (list_fd, 5);

        for (;;) {
            set_to_zero ((char *) &peer_sa, sizeof (peer_sa));
            addr_len = sizeof (peer_sa);
            peer_fd = accept (list_fd, (struct sockaddr *) &peer_sa, &addr_len);
            if (peer_fd < 0)
                continue;
STOP        pid = fork();
            if (!pid)
                break;
            close (peer_fd);
        }
        close (list_fd);
        return (peer_fd);
    }
```

```
Thread (1)
1/20391  T   in .listen_and_accept
```

```
Action Points
STOP  line  57 in .listen_and_accept+0xe8
```

Variable window

```
.listen_and_accept:list_sa (27252.1)
 (at 0x2ff21e88) Type: struct sockaddr_in

Field        Type             Value

sin_len      u_char           '\000' (0x00, or 0)
sin_family   u_char           '\002' (0x02, or 2)
sin_port     u_short          0x2694 (9876)
sin_addr     struct in_addr   (Compound Object)
   s_addr    in_addr_t        0x00000000 (0)
sin_zero     <string>[8]      ""
```

Figure 3. Sample TotalView Session

Figure 3 shows an example of a TotalView session containing the following windows:

Root Lists the name, location (if remote process), process ID, status, and optionally the list of threads for each process you are debugging. Lists the thread ID, status and current routine executing for each thread.

Process Displays information about a process and a thread within that process. Displays the stack trace, stack frame, and source code for the selected thread in a series of separate panes. Optionally displays disassembled machine code or interleaved source code and disassembled machine code.

Process Groups Displays the process groups for all of the multiprocess programs you are debugging.

Variable Displays the address, data type, and value of a local variable, register, or global variable. Also displays the values (and optionally, the machine-level instructions) stored in a block of memory.

The process window provides very detailed information about a process, including:

- The name, location (if remote process), process ID, and status of the process
- The name, location (if remote thread), thread ID, and status of the selected thread within the process
- The stack trace for the thread, with the selected routine highlighted
- The stack frame for the selected routine
- The source code for the selected frame (providing the routine was compiled with source line information) or disassembled machine code
- The current Program Counter (PC) for the selected stack frame, which is represented by an arrow on the line number of source code
- The breakpoints and evaluation points that are set in the source or machine code, as shown in the source pane
- The list of threads that exist within the process
- The list of breakpoints and evaluation points that are set in the process.

Examining Source and Machine Code

TotalView provides the following features for examining your code:

- Dive on functions

  When you dive on a function, its source code is displayed in the source code pane of the process window. See “Diving into Objects” on page 28 for more information.

- Search for functions

  You can search for functions using a dialog in the process window. See “Finding the Source Code for Functions” on page 116 for more information.

Controlling Processes and Threads

For controlling processes and threads, the TotalView debugger offers a full range of functions from the process window.

- Start and stop processes and threads

  You can start, stop, resume, delete, and restart your program. See page 124, page 127, and page 144 for information about how to perform these tasks.

- Attach to existing processes

  TotalView provides a window for examining processes that are not running under the debugger’s control. Attaching to one of these processes is as easy as diving on it. See “Attaching to Processes” on page 40 for more information.

- Examine core files

  When you start TotalView, you can load a core file and examine it in the same way as any executable. Or, you can load a core file anytime during a TotalView debugging session. See “Examining a Core File” on page 43 for more information.

- Change the way TotalView handles signals

  TotalView provides a dialog for tailoring how signals are handled. TotalView can stop the process and place it in the stopped state, stop the process and place it in the error state, send the signal on to the process, or discard the signal. See “Handling Signals” on page 48 for details.

- Single step

  You can single step through your program or step over function calls. You can also continue execution to a selected source line or instruction and continue execution until a function completes execution. TotalView supports process level, process group level, and on some systems, thread level single stepping. See “Single-Stepping” on page 133 for details.

- Reload the executable file

  After editing and recompiling a program, you can reload the executable file. See “Loading Executables” on page 38 for more information.

- Change the program counter (PC)

  You can change the value of the PC to resume execution at a different point in the program. See “Setting the Program Counter” on page 143 for more information.

Using Action Points

TotalView provides a broad range of action points: points in a program where you stop execution or evaluate an expression.

- Action points

  You can set, delete, enable, disable, suppress, and unsuppress the following kinds of action points at both the source level and machine level.

  - Breakpoints
  - Barrier breakpoints
  - Conditional breakpoints are breakpoints that occur only if a code fragment (expression) is satisfied

  - Evaluation points are points where a code fragment is evaluated

- Expressions and code fragments

  With the expression evaluation window and evaluation points, you can write and evaluate fragments of code, including function calls used by the current process. Depending on the platform, you can write fragments in C, C++, Fortran, or Assembler. On most platforms, TotalView interprets code fragments, but on some platforms, TotalView compiles the fragments.

  See Chapter 8, "Setting Action Points," on page 187 for more information about setting action points and writing evaluation expressions.

Examining and Manipulating Data

The TotalView debugger also offers a number of useful functions for examining and manipulating data in your program:

- Diving

  You can examine data by *diving* (clicking the right mouse button) into the variable or by issuing a command. You can examine local variables, registers, global variables, machine-level instructions, and areas of memory. In all cases, the debugger displays the information about the variable, register, or memory region in a separate variable window. See"Diving into Objects" on page 28 for more information.

- Changing values

  You can edit the value of a variable or a memory location to change it for the current running process. See "Changing the Values of Variables" on page 153 for more information.

- Changing types

  You can edit the type strings of variables to display the data in different formats. See "Changing the Data Type of Variables" on page 154 for more information.

- Laminated variables

  You can examine the value of a variable across multiple processes or multiple threads in a single data window. See “Displaying a Variable in All Processes or Threads” on page 177 for more information.

Visualizing Array Data

The TotalView debugger allows you to visualize array data in the programs you are debugging. This gives you an overall picture of your data and helps you to find incorrect data quickly and easily.

The Visualize program runs as a separate process, connecting to TotalView by a pipe. You interact with TotalView to choose *what* to visualize and *when* to update the images, and you interact with the Visualizer program to choose *how* to display the data.

Note: The Visualize program is not available on all platforms.

You can visualize array data in the following ways:

- **Visualize (v)** variable window menu item

  You can visualize the array data displayed in a variable window on demand by invoking the **Visualize (v)** menu item. This command gives you a visual snapshot of the array data listed in the window. Each time you visualize the same array data, the visualizer image is updated.

- **$visualize** expression system built-in statement

  You can use the **$visualize** expression system built-in statement in expressions called both from the expression evaluation window, and evaluation action points. The expression system allows you to visualize several different data-sets from a single expression. Each time the expression is evaluated, the set of images are automatically updated in the Visualizer program, allowing you to animate the visual representation of your data.

TotalView allows you to use your own visualization program. The data format generated by TotalView is described in a header file included with the TotalView distribution. For more information about visualization, see “Visualizing Data” on page 231.

Distributed Debugging

TotalView provides a distributed architecture that suits many different operating environments, including:

- Remote programs running on a separate machine from TotalView
- Distributed programs running on a set of homogeneous machines
- Multiprocess programs running on a multiprocessor machine
- Multiprocess programs running on a cluster of separate homogeneous machines
- Client-server programs with the server running on one machine type and the clients running on another machine type

Note: Distributed debugging currently requires that all machines have the same machine architecture and operating system

The machine on which TotalView is running is known as the *host machine*, while the machine on which the process being debugged is running is known as the *target machine*. When the host and target machines are the same, you can use TotalView as a native debugger. When the host and target machines are separate machines, you can use TotalView as a distributed debugger. When you use TotalView as a distributed debugger, it starts a process on each remote target machine. This process is called the TotalView Debugger Server (**tvdsvr**), and TotalView communicates with it using standard TCP/IP protocols (see Figure 4).

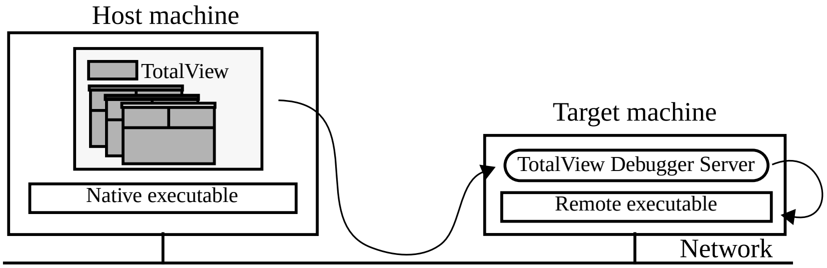


Figure 4. The TotalView Debugger Server

There are no differences in debugging distributed programs; TotalView offers the same set of rich features as with native programs and multiprocess programs.

In addition, on some platforms, TotalView can debug programs that use the HPF, MPI, IBM Parallel Environment (PE) or Parallel Virtual Machine (PVM) libraries, which are popular multiprocess programming libraries.

For more information on distributed debugging, refer to:

- “Debugging Remote Processes” on page 60
- “Debugging MPI Applications” on page 76
- “Debugging IBM MPI (PE) Applications” on page 82
- “Debugging PVM and DPVM Applications” on page 95
- “Debugging Portland Group, Inc. (PGI) HPF Applications” on page 103

Multiprocess Programs

The TotalView debugger has some special features for debugging multiprocess programs. Note that all of the user interface and debugging features that were discussed earlier in this chapter are also available for multiprocess programs.

- Separate windows for each process

  Each process has its own process window displaying information for that particular process. You can monitor the status, thread list, breakpoint list and source code, for *each* process in a multiprocess program. You don't have to display all the process windows in a multiprocess program; you can choose which process windows to open and close.

- Sharing of breakpoints among processes

  By setting options on the breakpoint in a parent process, you can control whether or not the breakpoint is shared among the child processes. You can also control whether or not all processes in the group stop when any process in the group reaches the breakpoint. See "Breakpoints for Multiple Processes" on page 197 for more information.

- Process groups

  The TotalView debugger treats multiprocess programs as process groups. If you debug several multiprocess programs at once, you can view information about all process groups. You also can view information about a particular multiprocess program by requesting information about its process group. You can start and stop an individual process group. See "Examining Process Groups" on page 129 for more information.

- Process barrier breakpoints

  In addition to "normal" breakpoints, TotalView allows you to create process barrier breakpoints. A process barrier breakpoint (process barrier point) is just like a normal breakpoint, but it holds processes that reach the barrier point until all the processes in the group reach it. When the last process in the group reaches a barrier, all processes in the group are released. While a process is held, attempts to continue the process do nothing. This is useful for synchronizing a group of processes at the same location. See "Process Barrier Breakpoints" on page 201 for more information.

- Process group-level single-stepping

  TotalView allows you single-step groups of processes with a single command. See "Group-Level Single-Stepping" on page 134 for more information.

- Single event log containing information for all processes

  The TotalView debugger logs significant events about each process you are debugging. Thus, you can view the history of your entire debugging session by scrolling through the event log window. See "Monitoring TotalView Sessions" on page 57 for more information.

- Automatically attach to child processes

  If a program calls **fork()** or **execve()**, TotalView automatically attaches to the child processes and includes them in the process group. See"Attaching to Processes" on page 40 and "Breakpoint for Programs that fork()/execve()" on page 199 for more information.

- Multiple symbol tables

  If you are debugging more than one executable at a time, TotalView automatically handles the symbol table for each executable

Multithreaded Programs

Most modern operating systems support running programs with multiple threads of execution. The implementation of threads varies among operating systems, but most thread implementations share the following characteristics:

- Shared address space

  The threads share an address space (memory) with other threads. They can read and write the same variables and can execute the same code.

- Private execution context

  Each thread has its own set of general-purpose registers and floating-point registers (if applicable to the processor).

- Private execution stack

  Each thread has a region of address space reserved for its execution stack. This is typically a range of addresses in the address space reserved for the thread's stack. However, one thread's stack can be read and written by other threads sharing the address space.

- Thread private data

  Some operating systems (not all) allow a program to "declare" thread private data. A program variable that is declared thread private provides each thread its own copy of the variable. Changes to the variable by one thread are not seen by the others. This facility usually requires compiler and linker support, in addition to operating system support.

TotalView supports debugging threaded applications on a variety of operating systems. On most versions of UNIX operating systems that support threads, a process consists of an address space and a list of one or more threads. Other operating systems that TotalView supports implement tasks or threads running in the memory space of a computer, and have no facilities for multiple processes or address spaces on a single machine.

To handle this diversity, TotalView implements a general model of address spaces and execution contexts. For conciseness, we use the term **thread** to mean a thread or task with an execution context, and **process** to mean an address space or computer memory that is capable of running one or more threads.

See "Navigating in the Process Window" on page 24 and "Determining the Status of Processes and Threads" on page 44 to learn how TotalView presents information about threads.

Context-Sensitive Help

You can request help from every window in the TotalView debugger. The **Help** command displays context-sensitive information about the window or dialog box you are currently working in or the debugging operation you are currently using. The debugger displays the information in a separate help window, so you can scroll through the information as you debug your program. As you make successive help requests, the debugger displays the new information in the help window. See "Getting Help" on page 19.

CHAPTER 2: TotalView Basics

This chapter introduces you to the TotalView interface. You'll learn how to:

- Compile your program
- Start TotalView
- Use the mouse buttons and menus
- Get online help
- Use the primary windows
- Scroll windows and fields
- Dive into objects
- Edit text
- Search for text strings
- Use the spelling corrector
- Save the contents of windows
- Exit TotalView

Compiling Programs

Before you start TotalView, compile your source code with the **–g** compiler switch, which generates debugging information in the symbol table. For example:

```
% cc –g program –o executable
```

For more information on compiling your program for TotalView, see “Compiling Programs” on page 36. On some platforms, additional compiler switches may be necessary or recommended for effective debugging. For more information, refer to Appendix A, ”Compilers and Environments,” on page 303.

If necessary, you can debug programs that have not been compiled with the **–g** compiler switch or programs for which you do not have the source code. For more information, refer to “Examining Source and Assembler Code” on page 120.

Starting TotalView

Depending on the kind of program you are debugging, there are several way to start TotalView. In its simplest form, use the **totalview** command with the name of your program (*filename*):

```
% totalview filename
```

For more information on starting TotalView, see “Starting the TotalView Debugger” on page 37.

For information on starting TotalView on a parallel debugging session, Chapter 5, “Setting Up Parallel Debugging Sessions,” on page 75.

For more information on the **totalview** command, command options, and command syntax, refer to Chapter 12, “TotalView Command Syntax,” on page 287.

Using the Mouse Buttons

The TotalView debugger supports a three-button mouse, as outlined in Table 1.

Table 1. Mouse Button Functions

Button	Default Position	Purpose	How to Use It
Select	Left	Select or edit object, scroll in windows and panes	Move the pointer over the object and click the button.
Menu	Middle	Display pop-up menu	Move the pointer into the window and hold down the button.
		Select command from menu	Move pointer down the menu until the desired command is highlighted, and release the button.
		Leave menu without selecting command	Move the pointer off the menu and release the button.
Dive	Right	Dive into object to display information about it	Move the pointer over the object and click the button.

In the tag field area (See Figure 7 on page 21 for an example of the tag field) of the source code pane, the select button has a special function. By selecting the line number of an executable line of code, you set a breakpoint at that line. TotalView displays a STOP sign in the tag field.

Selecting the STOP sign clears (deletes) the breakpoint. If an evaluation point has been set (indicated by an EVAL sign), selecting the sign disables it. For more information on breakpoints and evaluation points, refer to Chapter 8, “Setting Action Points,” on page 187.

Using Menu and Keyboard Commands

Each TotalView window provides a pop-up menu of commands for examining and manipulating the information displayed in a window. Figure 5 shows an example of the process window menu and a submenu. To display a pop-up menu in the current window, click the middle mouse button.

Many commands have keyboard shortcut (accelerator) keys that are shown in parentheses in this manual. For example, typing the letter **q** into the window issues the **Close Window (q)** command. The keyboard shortcuts are listed on the menu to the right of the menu command.

Items that appear dimmed on the menu are commands that are currently disabled.

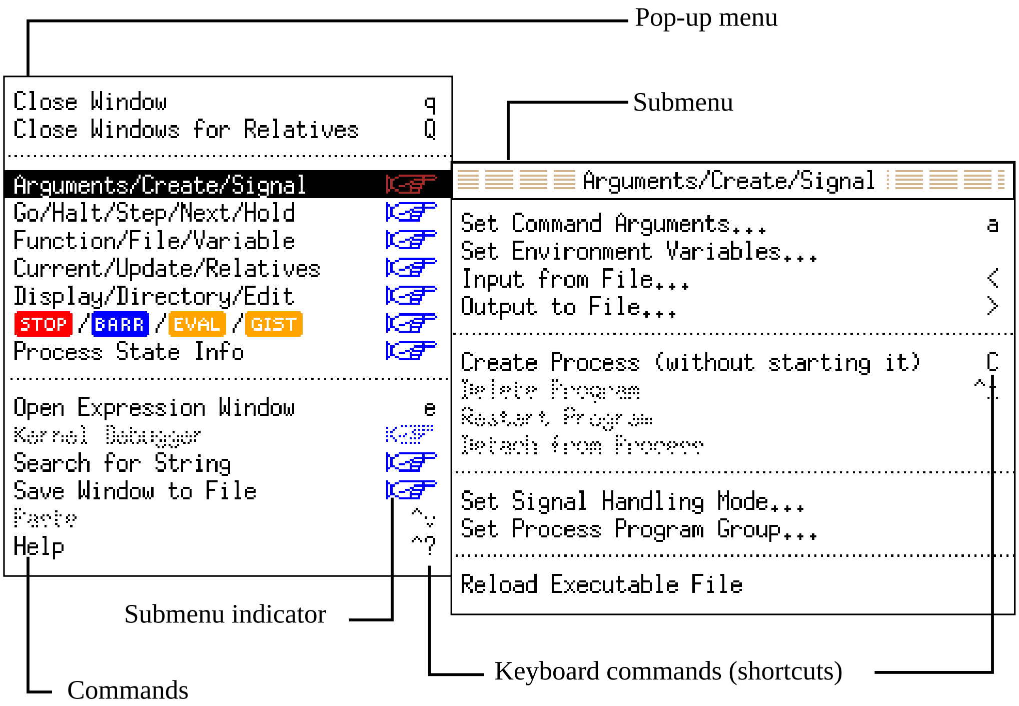


Figure 5. Example TotalView Menu and Submenu

The following commands are only available from the keyboard:

Control-C Cancels the single-step operation and other time-consuming operations, such as searching for a string.

Control-L Refreshes the current window.

Control-Q Exits from the debugger after you confirm.

Control-R Raises the root window.

Shift-Return Exits from the field editor that lets you to edit text in Totalview windows.

Getting Help

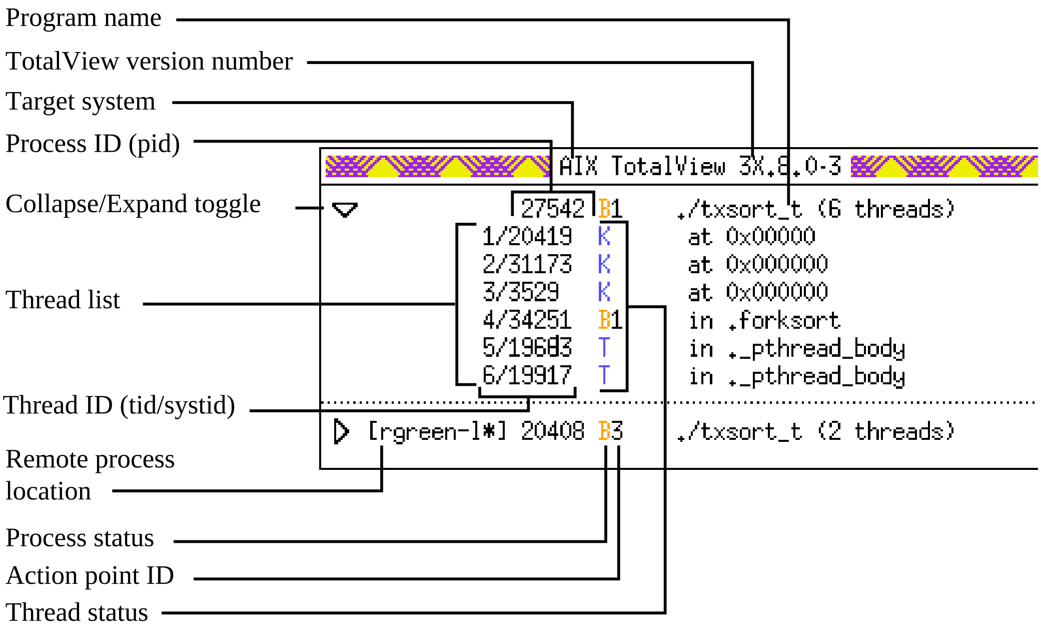
You can request help from any TotalView window or dialog box by selecting the **Help** command from the pop-up menu or by pressing **Control-?**. When you request help, a separate help window appears. To close the help window, select the **Close Window (q)** command from the menu.

Using the Primary Windows

When you start the TotalView debugger with the name of program to debug, two windows appear:

- The root window displays a list of all the processes that you are debugging, and optionally a list of thread for each process. Until you start a process, the root window lists only the name of the program with which you started TotalView.
- The process window displays the thread list, action point list, and the selected thread of a particular process that you are debugging. The process window also displays the source code, stack frame, and stack trace of the selected thread in that process. Until you start the process, the process window displays only the source code for the program.

Figure 6 and Figure 7 show the root and process windows.



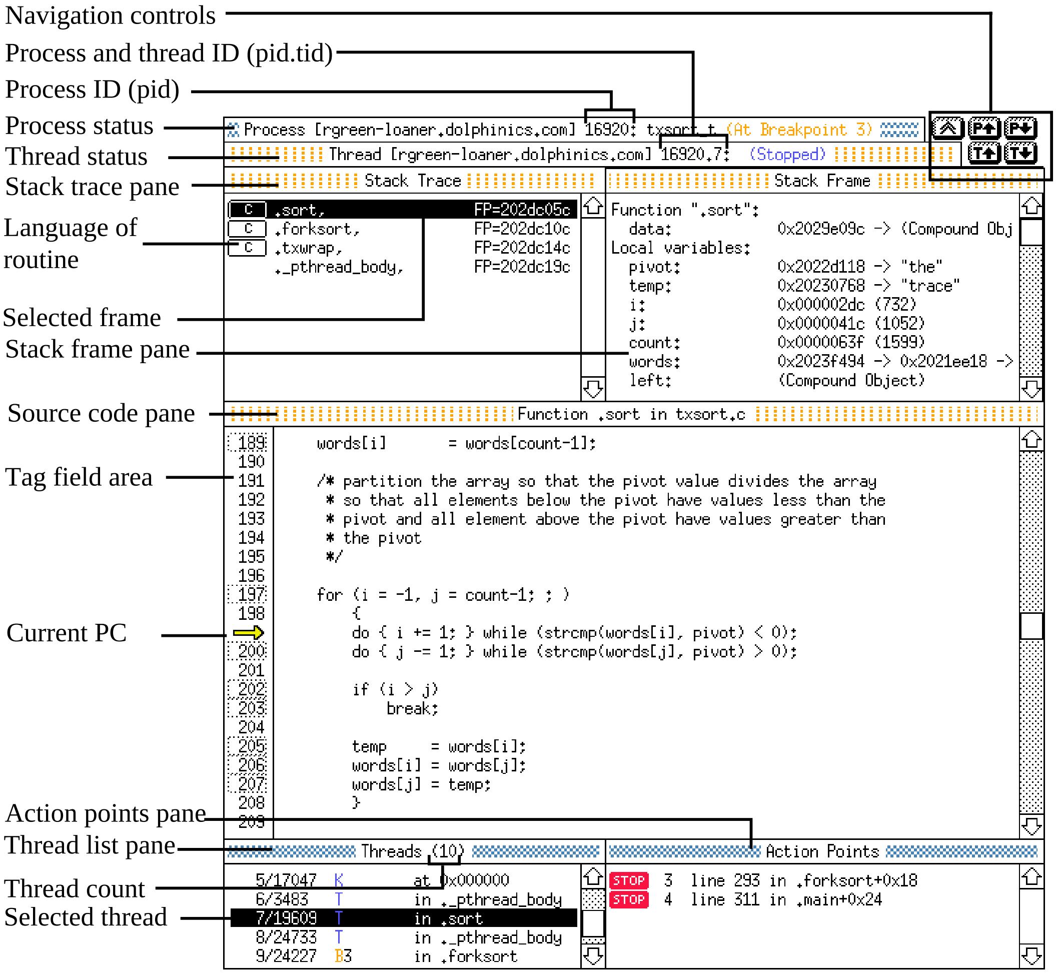
```
AIX TotalView 3X.8.0-3
▽          27542 B1    ./txsort_t (6 threads)
       1/20419   K      at 0x00000
       2/31173   K      at 0x000000
       3/3529    K      at 0x000000
       4/34251   B1     in .forksort
       5/19683   T      in ._pthread_body
       6/19917   T      in ._pthread_body
▷ [rgreen-l*] 20408 B3  ./txsort_t (2 threads)
```

Figure 6. Root Window

Starting A Process

To start a process:

1. Move your cursor to the process window.
2. Set a breakpoint in the source code by selecting a boxed line number.
3. Type the keyboard accelerator **g (**for the **Go Process** command). The process starts running and then stops at the breakpoint you set.



```
Process [rgreen-loaner.dolphinics.com] 16920: txsort_t (At Breakpoint 3)
Thread [rgreen-loaner.dolphinics.com] 16920.7: (Stopped)
```

```
Stack Trace
C   .sort,                  FP=202dc05c
C   .forksort,              FP=202dc10c
C   .txwrap,                FP=202dc14c
    ._pthread_body,         FP=202dc19c
```

```
Stack Frame
Function ".sort":
  data:              0x2029e09c -> (Compound Obj
Local variables:
  pivot:             0x2022d118 -> "the"
  temp:              0x20230768 -> "trace"
  i:                 0x000002dc (732)
  j:                 0x0000041c (1052)
  count:             0x0000063f (1599)
  words:             0x2023f494 -> 0x2021ee18 ->
  left:              (Compound Object)
```

```
Function .sort in txsort.c
    words[i]       = words[count-1];

    /* partition the array so that the pivot value divides the array
     * so that all elements below the pivot have values less than the
     * pivot and all element above the pivot have values greater than
     * the pivot
     */

    for (i = -1, j = count-1; ; )
        {
        do { i += 1; } while (strcmp(words[i], pivot) < 0);
        do { j -= 1; } while (strcmp(words[j], pivot) > 0);

        if (i > j)
            break;

        temp      = words[i];
        words[i] = words[j];
        words[j] = temp;
        }
```

```
Threads (10)
   5/17047   K          at 0x000000
   6/3483    T          in ._pthread_body
   7/19609   T          in .sort
   8/24733   T          in ._pthread_body
   9/24227   B3         in .forksort
```

```
Action Points
STOP  3  line 293 in .forksort+0x18
STOP  4  line 311 in .main+0x24
```

Figure 7. Process Window

When you are debugging a remote process the abbreviated hostname on which the process is running appears in square brackets in the root window, and the full hostname appears in square brackets in the title bar of the process window. For example, in Figure 7, the process running **txsort_t** is on the machine **rgreen-loaner.dolphinics.com**, which is abbreviated to **[rgreen-l*]** in the root window. In the process window, the full hostname of the process **[rgreen-loaner.dolphinics.com]** is displayed.

As you examine the process window in Figure 7, notice the following:

- The thread list pane shows the list of threads that currently exist in the process. The number in the thread list pane title is the count of the number of threads that currently exist in the process. When you select a different thread in the thread list, TotalView updates the stack trace pane, stack frame pane, and source code pane to show you the information for that thread. When you dive on a different thread in the thread list, TotalView finds or opens a new window displaying information for that thread. Holding down the Shift key when you dive will force TotalView to open a new process window focused on that thread.
- The thread ID shown in the root window and thread list pane of the process window is in the format *tid*/*systid*. *tid* is the TotalView assigned logical thread ID and *systid* is the system assigned thread ID. On systems such as Digital UNIX, where the *tid* and *systid* values are the same, TotalView displays only the *tid* value.
- In other windows, TotalView uses *pid***.***tid* to identify threads within a process.
- The stack trace pane shows the call stack of routines that are executed by the selected thread. You can move up and down the call stack by selecting the desired routine (stack frame). When you select a different stack frame in the call stack, TotalView updates the stack frame pane and source code pane to show the information about the selected routine.
- The stack frame pane displays all the function parameters, local variables, and registers for the selected stack frame.
- The information displayed in the stack trace and stack frame panes reflects the state of the process when it was last stopped. Therefore, the stack trace and stack frame panes are not current while the thread is running.
- In the left margin of the source code pane, the tag field area contains line numbers opposite all lines of source code. You can place a breakpoint at any line of source code that generated object code, which is indicated by a boxed line number. The arrow in the tag field indicates the current location of the

program counter (PC) for the selected stack frame. See Figure 8 for more information.

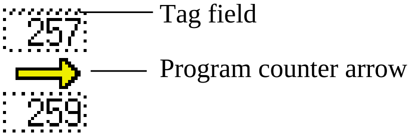


Figure 8. Program Counter

- In a multiprocess or multithreaded program, each thread has its own point of execution, so the program counter arrow points to a unique program counter (PC) in each process window for a particular thread. Therefore, when you stop a multiprocess or multithreaded program, the routine selected in the stack trace pane for a particular thread depends on the PC for the thread. At the time you stop the program, some threads might be executing in one routine, while others might be executing in other routines.
- The action points list pane shows the list of breakpoints and evaluation points for the process.
- The navigation control buttons in the upper right-hand corner of the process window allow you to easily navigate through the processes and threads you are debugging.

Sizing Process Window Panes

You can change the size the panes in the process window. If you do not want to see a particular pane, you can size the pane to a zero size. To do so:

1. Move the mouse cursor over the edge of the window pane until the cursor with crossed arrows appears, as shown in Figure 9:

Figure 9. The Sizing Cursor

2. Hold down the left mouse button and drag the edge until the pane is the desired size.

Navigating in the Process Window

The navigation control buttons, located in the upper right corner of the process window, allow you to easily navigate through the processes and threads you are debugging. Using these buttons you can:

- Move up and down the list of processes you are debugging
- Move up and down the list of threads in a particular process
- Go back to the previous contents of the process window

Figure 10 shows the navigation controls available in the process window.

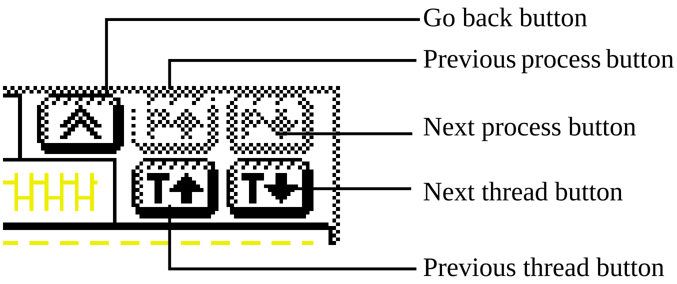


Figure 10. Process Window Navigation Controls

Navigating in the Root Window

You can also navigate through the processes and threads you are debugging from the root window. In general, selecting a process or thread with the left mouse button will not open a new window. Selecting tries to minimize the number of open process windows. However, diving on a process or thread with the right mouse button will open a new process window if an exactly matching process/thread combination could not be found. Finally, holding down the Shift key when you dive always opens a new window.

- When you select a process in the root window, TotalView finds or opens a process window for that process. If a matching window can’t be found, it will replace the contents of an existing process window and show you the selected process.
- When you dive on a process in the root window, TotalView finds or opens a process window for that process. Holding down the Shift key when you dive will force TotalView to open a new process window focused on that process

- When you select a thread in the root window, TotalView finds or opens a process window for that process and show you the selected thread. If a matching window can't be found, it will replace the contents of an existing process window and show you the selected thread.
- When you dive on a thread in the root window, TotalView finds or opens a process window for that process and thread combination. Holding down the Shift key when you dive will force TotalView to open a new process window focused on that thread.

The Process Window Stack

Whenever the process and/or thread is replaced in the process window, the previous contents of the window are *pushed* onto a stack. The go back button *pops* the stack and shows you the previous contents of the process window. The process window stack is pushed in the following cases:

- Select or dive in the thread list pane in the process window
- Select or dive on any of the four process/thread previous/next buttons in the process window
- A select operation in the root window on a process or thread that causes the contents of a process window to be replaced with the selected process or thread.

Scrolling Windows and Fields

Scrolling Windows

You can use the scroll bars to scroll through the information in TotalView windows and panes, as shown in Figure 11.

- To scroll one line at a time, click the Select mouse button on the up or down arrows (at the top and bottom of the scroll bar).
- To scroll one page at a time, click the Select mouse button above or below the elevator box inside the scroll bar.
- To scroll an arbitrary amount, hold down the Select mouse button and drag the elevator box inside the scroll bar.

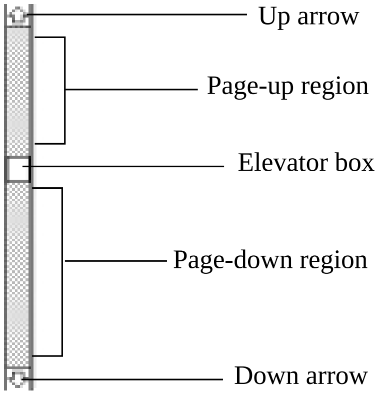


Figure 11. Scroll Bar

To scroll continuously by line or by page, you can hold down the Select mouse button instead of clicking it. If TotalView scrolls too fast or too slow, you can adjust the scrolling speed using X resources. Refer to “totalview*scrollLineSpeed: n” on page 276 for further information.

You can also scroll windows using the keys on your keyboard’s numeric keypad:

↑	Scrolls up one line.
Meta-↑	Scrolls up one page.
↓	Scrolls down one line.
Meta-↓	Scrolls down one page.
Page up	Scrolls up one page.
Page down	Scrolls down one page.

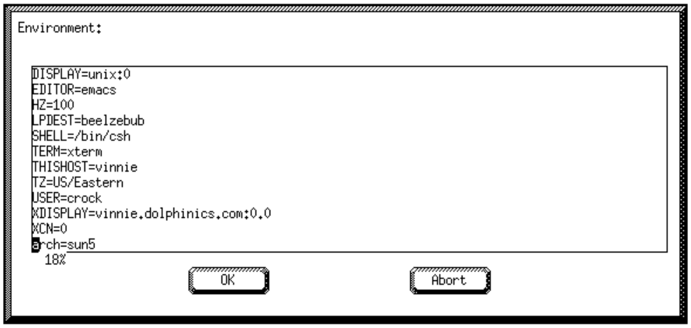
On some platforms, you may need to adjust your X Window System keyboard mapping to use certain keys on the numeric keypad. Refer to Appendix B, ”Operating Systems,” on page 321 for details.

Scrolling Multiline Fields

You can scroll multiline fields in dialog boxes, which allows you to create more lines than are visible. The bottom left corner of the multiline field indicates your location in the field with the following symbols:

- **All** – All of the lines in the field are visible.
- **Top** – The top line of the field is visible, but there are more lines below the bottom of the field that are not visible.
- **Bot** – The bottom line of the field is visible, but there are more lines above the top of the field that are not visible.
- *nn***%** – The percentage of the lines above the top of the field that are not visible.

Figure 12 shows an example of a scrollable multiline field.



Environment:

```
DISPLAY=unix:0
EDITOR=emacs
HZ=100
LPDEST=beelzebub
SHELL=/bin/csh
TERM=xterm
THISHOST=vinnie
TZ=US/Eastern
USER=crock
XDISPLAY=vinnie.dolphinics.com:0.0
XCN=0
arch=sun5
  18%
```

Figure 12. Scrollable Multiline Field

You can use the ↑ key or **Control-P** to move up a line in a multiline field. You can also use the ↓ key or **Control-N** to move down a line in a multiline field. When you move off the top or bottom of the field and there are more lines above or below, the field scrolls automatically by one line.

You can scroll a multiline field by more than one line at a time by combining **Control-U** with any of the other commands for moving up or down a line. When you precede an editing command with **Control-U**, it repeats the command four times. For example, if you enter **Control-U Control-P**, the cursor moves up four lines.

Diving into Objects

To display more detail about an object (for example, a variable), dive into it by clicking the Dive mouse button. You can dive into any object that has a block of data associated with it, such as a pointer, structure, or subroutine. TotalView displays the information about the object in the current window or in a separate window, as outlined in Table 2.

Table 2. Uses for Diving

Object	**Information Displayed by Diving**
Process or thread	A process window appears focused on a thread. See “Using the Primary Windows” on page 20.
Routine in the stack trace pane	The stack frame and source code for the routine appear in the process window.
Pointer	The referenced memory area appears in a separate variable window.
Variable	The contents of the variable appears in a separate variable window.
Array element, structure element, or referenced memory area	The contents of the element or memory area replaces the contents that was in the variable window. This is known as a *nested* dive.
Subroutine[1]	The source code for the subroutine appears in the process window.

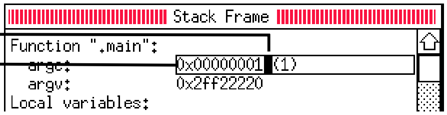
[1] A subroutine must be compiled with source line information (usually, with the **–g** switch) for you to dive into it and see source code. If the subroutine was not compiled with source line information, the debugger displays the assembler code for the routine.

For additional information about displaying variable contents, refer to “Diving in Variable Windows” on page 152.

Editing Text

To change the values of fields in TotalView windows, or to change text fields in dialogs, you can use the field editor, which has basic text editing capabilities. To edit text:

1. Click the left mouse button to select the text to change.
2. If you can edit the selected text, it is enclosed in a rectangle, and the ***editing cursor*** (a black rectangle) appears, as shown in Figure 13.



Editing cursor

Selection box

```
Stack Frame
Function ".main":
  argc:        0x00000001 (1)
  argv:        0x2ff22220
Local variables:
```

Figure 13. Editing Cursor

3. Edit the text and press Return (for single-line fields) or Shift-Return (for multiline fields).

You can copy and paste text within TotalView windows, between TotalView windows, or between TotalView windows and other X Window System windows.

To copy and paste text between an editable field in TotalView and other X windows applications, do the following:

1. Copy text into the cut buffer with one of the following:

   - Click and hold the left mouse button at one end of the range, drag the cursor to the other end of the range, then let go of the mouse button; or
   - Click the left mouse button at one end of the range then click right mouse button at the other end of the range

   TotalView highlights the text while you hold the mouse button down. When you release the mouse button, the highlight disappears indicating TotalView copied the text into the cut buffer.

2. Move the cursor to the place you want to paste the text, then do one of the following:

- Press Control middle mouse button; or
- Press the middle mouse button for a menu. Select **Paste (Control-V)** from the menu.

Note: The preceding steps apply to copy and paste operations for TotalView windows only, not to other X Window System clients.

The field editor supports some of the same commands as GNU Emacs, as outlined in Table 3.

Table 3. Field Editor Commands

Keystrokes	**Action**
Control-A	Move the cursor to the beginning of the line.
Control-B	Move the cursor backward one character.
Control-C	Abort the field editor, and discard all changes.
Control-D	Delete the character under the cursor.
Control-E	Move the cursor to the end of the line.
Control-F	Move the cursor forward one character.
Control-H, Backspace, or Delete	Delete the previous character.
Control-K	Delete all text to the end of the line, or delete a newline.
Control-N	Move the cursor to the next line (in fields with multiple lines only).
Control-O	Insert a newline (in fields with multiple lines only).
Control-P	Move the cursor to the previous line (in fields with multiple lines only).
Control-U [*n*]	Multiply the number of times the command is executed by *n*. *n* is optional; the default is 4. Issue this command in combination with another command. For example, to move the cursor forward 50 characters, you enter: Control-U 50 Control-F.

Table 3. Field Editor Commands (Continued)

Keystrokes	Action
Control-V	Paste text from X windows copy buffer.
Tab	Space over to the next tab stop. (Tab stops are located every four characters.)
Return	For single-line fields, stop the field editor and deselect the field. In dialog boxes, confirm the dialog box as if the **OK**, **Continue** or **Yes** button was selected. For multi-line fields, insert a newline.
Shift-Return	For both single-line and multi-line fields, stop the field editor and deselect the field. In dialog boxes, confirm the dialog box as if the **OK**, **Continue** or **Yes** button was selected.
↑, ↓, ←, →	Move up, down, backward, and forward one character.

Searching for Text

You can search for text strings in most TotalView windows. You can use the following commands:

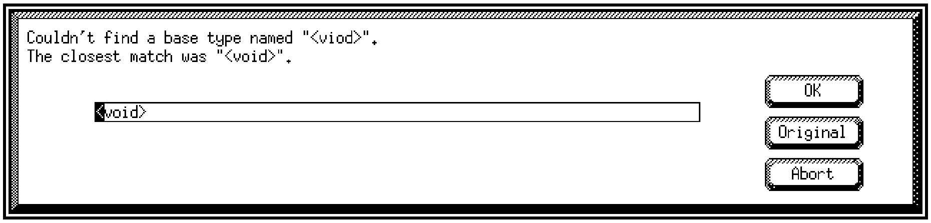
Search for String(/) Searches forward in the window for a text string. The debugger prompts you for the string. The search starts from the first line of text that is visible in the window.

Search Backward for String (\) Searches backward in the window for a text string. The search starts from the last line of text that is visible in the window.

Reexecute Last Search (.) Repeats the last forward or backward search without prompting for a string. The search starts from the point where the last search left off and continues in the same direction.

Using the Spelling Corrector

TotalView checks the spelling of text entries for certain commands. If TotalView does not find the name you entered, it displays a dialog box with the closest match, as shown in Figure 14.



Couldn't find a base type named "<viod>".
The closest match was "<void>".

<void>

Figure 14. Dialog Box for Spelling Corrector

You can edit the closest match, and then select **OK** to use it, **Original** to get back the original text, or **Abort** to cancel.

To customize the behavior of the spelling corrector with X Window System resources, refer to “totalview*spellCorrection: {verbose | brief | none}” on page 280.

Saving the Contents of Windows

You can save the contents of most window panes as ASCII text. You can save the contents in the following ways:

- Write it to a file. When you specify *filename*, TotalView creates the file (if it does not exist) and overwrites its contents with the text.
- Append it to a file. When you specify +*filename*, TotalView creates the file (if it does not exist) and appends the text to the end of it.
- Pipe it to UNIX shell commands. When you specify |*command*, TotalView pipes the commands to **/bin/sh** for execution. You can use a series of complex shell commands if desired. For example, to ignore the top five lines of output,

compare the current ASCII text to an existing file, and write the differences to another file, you specify:

|tail +5 | diff – *filename* > *filename.diff*

To save the contents of the current window pane:

1. Move the mouse pointer into the desired pane.
2. Select the **Save Window to File** command.
3. Enter *filename*, **+***filename*, or **|***command* in the dialog box.
4. Press Return.

To save a series of panes in a window, you can use the **Reexecute Last Save Window** command. This command repeats the last **Save Window to File** command (including the information entered in the dialog box) but for the current window pane.

Exiting from the TotalView Debugger

You can exit from the debugger in two different ways:

- Press **Control-Q** in any window.
- Select the **Quit Debugger (q)** command in the root window.

In the dialog box, select **Yes** or type **y** to confirm. To cancel the exit, select **No** or type **n**.

CHAPTER 3: Setting Up a Debugging Session

This chapter explains how to set up basic TotalView sessions. It also describes how to implement some common commands and procedures. For information on setting up remote debugging sessions, see Chapter 4, “Setting Up Remote Debugging Sessions,” on page 59. For information on setting up parallel debugging sessions, see Chapter 5, “Setting Up Parallel Debugging Sessions,” on page 75.

In this chapter, you will learn how to:

- Compile programs
- Start TotalView
- Load executables
- Attach to and detach from processes
- Examine core files
- Determine the status of processes and threads
- Handle signals
- Set search paths
- Set command arguments and environment variables
- Set input and output files
- Monitor your TotalView session

Compiling Programs

Before you start to debug a program with the TotalView debugger, you must compile the program with the appropriate switches and libraries for your situation. Table 4 discusses some general considerations, but you must check Appendix A, “Compilers and Environments,” on page 303 to determine the exact syntax and any other considerations for your platform. For additional information on how to compile a Portland Group HPF program for debugging, see “Compiling HPF for Debugging” on page 106.

Table 4. Compiler Considerations

Compiler Switch or Library	What It Does	When to Use It
Debugging symbols switch (usually **–g**)	Generates debugging information in the symbol table	Before debugging *any* program with TotalView
Optimization switch (usually **–O**)	Moves code around to optimize execution of program [1]	After you finish debugging your program with TotalView
Multiprocess programming library (usually **dbfork**) [2]	Uses special versions of the **fork()** and **execve()** system calls	Before debugging a multiprocess program that explicitly calls **fork()** or **execve()** [3]

[1] Some compilers don’t permit you to use the **–O** switch simultaneously with the **–g** switch. Even if your compiler does permit this, we recommend against it. Although you can do some debugging with the **–O** option on, your debugging session may produce strange results.

[2] The TotalView **dbfork** library is distributed as two separate libraries on IRIX6 MIPS. Use the **libdfork_n32.a** library to link to **–n32** compiled executables. Use the **libdbfork_n64.a** library to link to **–64** executables.

[3] Refer to “Processes That Call fork()” on page 199 and “Processes That Call execve()” on page 199.

Starting the TotalView Debugger

The complete command syntax for starting the TotalView debugger is as follows:

```
% totalview [filename [corefile]] [options]
```

where *filename* specifies the name of the executable file to be debugged and *corefile* specifies the name of the core file to be debugged.

Here are some of the most common ways of starting the debugger:

totalview

Starts the debugger without loading a program or core file. Once in TotalView, you can load a program by issuing the **New Program Window (n)** command from the root window.

totalview *filename*

Starts the debugger and loads the program specified by *filename*.

totalview *filename corefile*

Starts the debugger and loads the program specified by *filename* and the core file specified by *corefile*.

totalview *filename* **–a** *args*

Starts the debugger and passes all subsequent arguments (specified by *args*) to the program specified by *filename*. The **–a** option must appear after all other TotalView options on the command line.

totalview *filename* **–grab**

Starts the debugger and grabs the keyboard whenever it displays a dialog box. You should use this option whenever you start TotalView with a window manager that uses a “click-to-type” model.

totalview *filename* **–remote** *hostname*[**:***portnumber*]

Starts TotalView on the local host and the TotalView Debugger Server (**tvdsvr**) on the remote host *hostname*. Loads the program specified by *filename* for remote debugging. You can specify a host name or TCP/IP address for *hostname*, and optionally, a TCP/IP port number for *portnumber*.

For more information on:

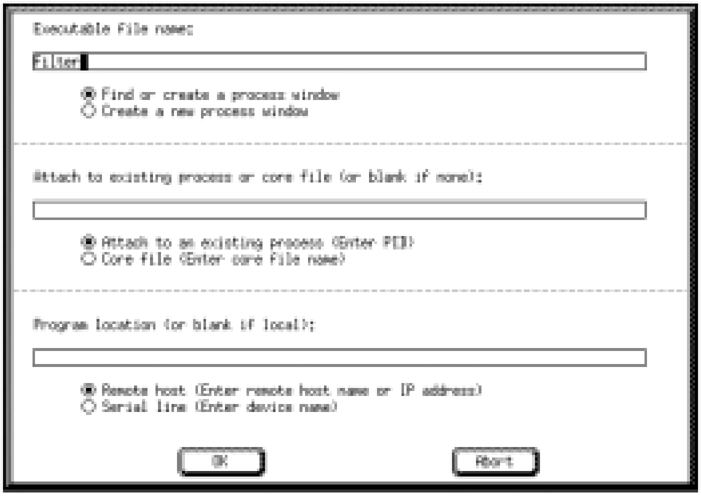
- debugging parallel programs such as MPI, PVM, or HPF, refer to Chapter 5, “Setting Up Parallel Debugging Sessions,” on page 75.
- the **totalview** command, refer to Chapter 12, “TotalView Command Syntax,” on page 287;
- remote debugging, refer to “Debugging Remote Processes” on page 60, “Starting the Debugger Server for Remote Debugging” on page 64, and Chapter 13, “TotalView Debugger Server Command Syntax,” on page 299;

Loading Executables

Loading a New Executable

If you did not load an executable when starting TotalView, you can load one at any time using the **New Program Window** command. To do so, do the following:

1. From the root window, select the **New Program Window (n)** command. A dialog box appears, as shown in Figure 15.



Executable file name:

Filter

● Find or create a process window
○ Create a new process window

Attach to existing process or core file (or blank if none):

● Attach to an existing process (Enter PID)
○ Core file (Enter core file name)

Program location (or blank if local):

● Remote host (Enter remote host name or IP address)
○ Serial line (Enter device name)

Figure 15. New Program Window Dialog Box

2. Enter the name of the executable in the top section of the dialog box. The name can be a full or relative pathname.

   If you supply a simple filename, TotalView searches for it in the list of directories specified with the **Set Search Directory** command and specified by your PATH environment variable.

3. (Optional) If you prefer to create a brand new process instead of reusing an existing one (the default), select the **Create a new process window** radio button.When you select this option, TotalView creates a new entry in the root window for the process
4. Press Return.

Note: If you use the **New Program Window** command to load the same executable again, TotalView does not reread the executable, and it reuses the existing symbol table. To have TotalView reread the executable, you need to use the **Reload Executable File** command, as described in the next section.

Reloading a Recompiled Executable

If you have edited and recompiled your program during a debugging session, you can reload your updated program without exiting from the debugger. To do so:

1. Confirm that all processes using the executable have exited. If they have not, display the **Arguments/Create/Signal** submenu and select the **Delete Program (^Z)** command from the process window.
2. Confirm that duplicate copies of the process do not exist by issuing the **ps** command in a shell. If duplicate processes exist, delete them with the **kill** command.
3. Recompile your program.
4. In the process window, display the **Arguments/Create/Signal** submenu and select the **Reload Executable File** command. The debugger updates the process window with the new source file and loads a new executable file. The next time you start the process, the debugger uses the new executable file.

Attaching to Processes

If a program you are testing is hung or looping (or misbehaving in some other way), you can attach to it with TotalView. You can attach to single processes, multiprocess programs, and remote processes.

To attach to a process, you can either use the **Show All Unattached Processes (N)** or **New Program Window (n)** commands.

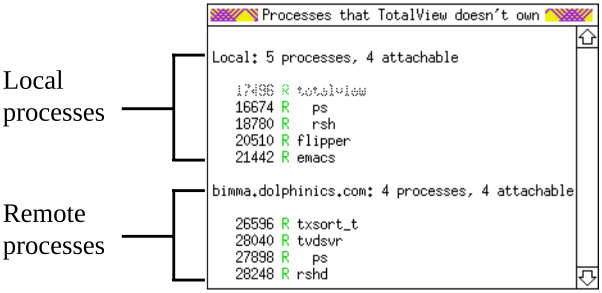
Note: If the process or any of its children has called the **execve()** routine, you may need to attach to it by creating a new program window. The reason for this is that on some platforms TotalView uses the **ps** command to obtain the name of the executable file for the process. Since **ps** can give incorrect names, TotalView might not be able to find the executable for the process.

Attaching Using Show All Unattached Processes

To attach to a process using the **Show All Unattached Processes (N)** command, go to the root window and complete the following steps:

1. Select the **Show All Unattached Processes (N)** command.

   The unattached processes window appears, as shown in Figure 16. This window lists the process ID, status, and name of each process associated with your username. The processes that appear dimmed are those that are already being debugged by the debugger, or those that TotalView will not allow you to debug (e.g., the TotalView process itself).



```
Local: 5 processes, 4 attachable

   17496 R totalview
   16674 R   ps
   18780 R   rsh
   20510 R flipper
   21442 R emacs

bimma.dolphinics.com: 4 processes, 4 attachable

   26596 R txsort_t
   28040 R tvdsvr
   27898 R   ps
   28248 R rshd
```

Figure 16. Unattached Processes Window

If you are debugging a remote process in this session, the unattached processes window also shows processes running under your username on each remote host name. You can attach to any remote process listed. For more information on remote debugging, refer to “Starting the Debugger Server for Remote Debugging” on page 64 and Chapter 13, “TotalView Debugger Server Command Syntax,” on page 299.

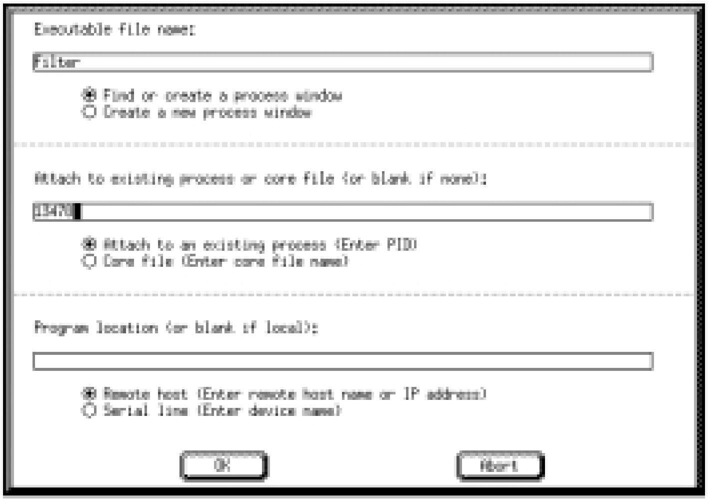
2. Dive into the process you wish to debug.

   A process window appears. The right arrow points to the current program counter (PC) The PC indicates where the program is either executing or hung.

Attaching Using New Program Window

To attach to a process with the **New Program Window (n)** command, follow these steps:

1. Get the process ID (PID) of the process by using the **ps** command in a shell.
2. Issue the **New Program Window (n)** command from the root window. A dialog box appears, as shown in Figure 17.



Executable file name:

filter

● Find or create a process window
○ Create a new process window

Attach to existing process or core file (or blank if none):

13470

● Attach to an existing process (Enter PID)
○ Core file (Enter core file name)

Program location (or blank if local):

● Remote host (Enter remote host name or IP address)
○ Serial line (Enter device name)

Figure 17. New Program Window Dialog Box

3. Enter the name of the executable in the top section of the dialog box. The name can be a full or relative pathname. If you supply a simple filename, TotalView searches for it in the list of directories specified with the **Set Search Directory** command and specified by your PATH environment variable.
4. Enter the process ID (PID) of the *unattached* process in the middle section of the dialog box.
5. Press Return.

   If the executable is a multiprocess program, the debugger asks you if you want to attach to all relatives of the process. If you want to examine all processes, select **Yes**.

If the process has children that called **execve()**, the debugger tries to determine the correct executable for each of them. If the debugger cannot determine the executables for the children, you need to delete (kill) the parent process and start it again using TotalView.

Finally, a process window appears. The right arrow points to the current program counter (PC). This is where the program is either executing or hung.

Detaching from Processes

You can detach from any processes to which you have attached (that is, processes that TotalView did not create) when you finish debugging them. When you detach from a process, TotalView removes all breakpoints that you set in that process.

To detach from a process:

1. If you want to send the process a signal, select the **Set Continuation Signal** command. Choose the signal that TotalView should send to the process when it detaches from it. For example, to detach from a process and leave it stopped, set the continuation signal to SIGSTOP.
2. Display the **Arguments/Create/Signal** submenu and select the **Detach from Process** command.

Examining a Core File

If a process encounters a serious error and dumps a core file, you can examine it from the debugger. TotalView provides two different methods for examining a core file:

- You can start the TotalView debugger with the following command:

  % **totalview** *filename corefile* [*options*]

  where *corefile* is the name of the core file.

- You can issue the **New Program Window (n)** command from the root window. In the dialog box, enter the name of the core file in the middle section of the dialog, select the **Core file** radio button, and press Return.

Note: You can debug only local core files. TotalView does not support remote debugging of core files.

The process window displays the core file, with the stack trace, stack frame, and source code panes showing the state of the process when it dumped core. The title bar of the process window specifies the signal that caused the core dump. The right arrow in the tag field of the source code pane indicates the value of the program counter (PC) when the process encountered the error.

You can examine all of the variables to see their state at the time the process found the error. For more information on examining variables, refer to Chapter 7, "Examining and Changing Data," on page 147.

If you start a process while you are examining a core file, the debugger stops using the core file and starts a fresh process with the executable.

Determining the Status of Processes and Threads

Process and thread states are displayed in:

- The root window, for processes and threads
- The unattached processes window, for processes
- The process and thread status bars of the process window, for processes and threads
- The thread list pane of the process window, for threads

Process Status

The status of a process includes three things: the process location, the process ID, and the state of the process. The root window displays a single character to identify the state of a process. The process status in the root window takes the following form:

[*L***]** *N S process_name*

where **[***L***]** is the process location (present only for remote processes), *N* is the process ID, *S* is the single-character representation of the process state, and *process_name* is TotalView's name for the process.

The unattached processes window lists all processes that are associated with your username. The format of the information in the unattached process window is similar to the format of processes in the root window. Process states are specified with a single character. Processes which you are debugging in your TotalView session are dimmed out.

The process status bar of the process window displays information in the following format:

Process [*L***]** *N***:** *process_name* **(***state***)**

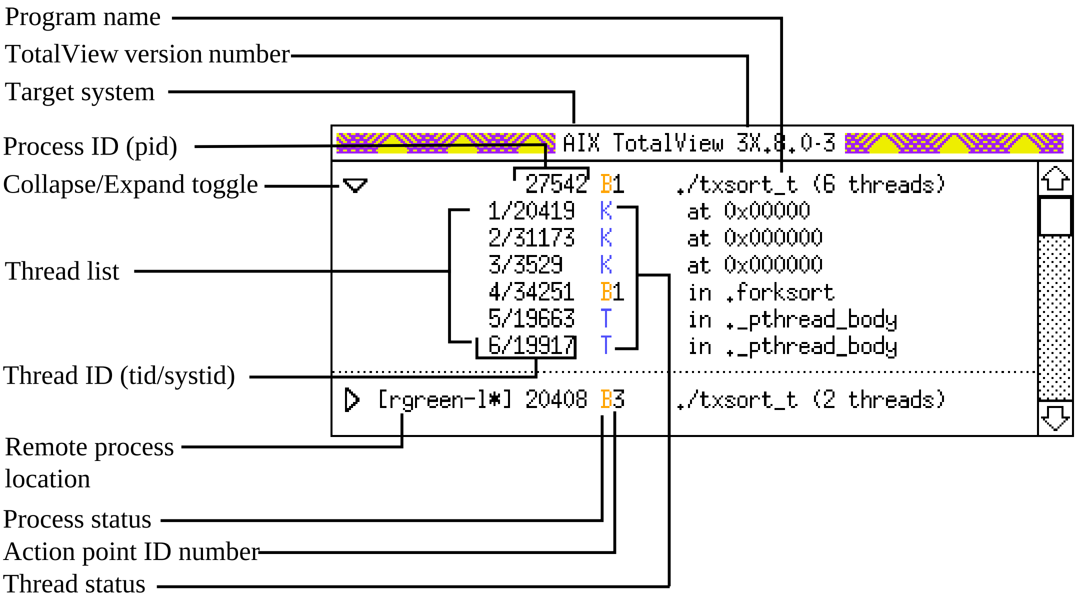
where **[***L***]** is the process location (present only for remote processes), *N* is the process ID, *process_name* is TotalView's name for the process, and *state* is the state name of the process based on the state of its threads.

Thread Status

The root window displays a single character to identify the state of a thread. The thread status in the root window takes the following form:

T/*X S* **in** *routine_name*

where *T* is the TotalView assigned thread ID, *X* is the system assigned thread ID, and *S* is the single-character representation of the thread state, and *routine_name* is the name of the routine in which the thread was executing when last stopped by TotalView. On systems for which the TotalView-assigned thread ID and the system-assigned thread ID are the same, TotalView displays only one ID value. See Figure 18.



```
AIX TotalView 3X.8.0-3
▽           27542  B1    ./txsort_t (6 threads)
          1/20419  K      at 0x00000
          2/31173  K      at 0x000000
          3/3529   K      at 0x000000
          4/34251  B1     in .forksort
          5/19663  T      in ._pthread_body
          6/19917  T      in ._pthread_body
▷ [rgreen-1*] 20408 B3   ./txsort_t (2 threads)
```

Figure 18. Root Window Showing Process and Thread Status

The thread list pane in the process window uses the same thread status format as the root window.

The thread status bar of the process window displays information in the following format:

> **Thread** *N*.*T*: *process_name* **(***state***)** <*reason*>

where *N* is the process ID, *T* is the TotalView assigned thread ID, *process_name* is TotalView's name for the process, *state* is the state name of the thread, and <*reason*> is the reason the thread stopped.

Unattached Process States

The state information for a process displayed in the unattached processes window is derived from the system. The state characters TotalView uses to summarize the state of an unattached process do not necessarily match those used by the system.

Table 5 summarizes the possible states in the unattached processes window.

Table 5. Summary of Unattached Process States

State	State Character	Meaning for a process
Running	**R**	Process is running or can run.
Stopped	**T**	Process is stopped.
Idle	**I**	Process has been idle or sleeping for *more* than 20 seconds.
Sleeping	**S**	Process has been idle or sleeping for *less* than 20 seconds.
Zombie	**Z**	Process is a “zombie,” a child process that has terminated and is waiting for its parent process to gather its status.

Attached Process States

The state of processes and threads that TotalView is attached to is displayed in various windows.

Table 6 summarizes the possible states for an attached process or thread, and how the states are displayed.

Table 6. Summary of Attached Process and Thread States

State Name	State Character	Meaning for a thread and process
Exited or never created	Blank	*Process only*: does not exist.
Running	**R**	*Thread*: is running or can run. *Process*: all threads in the process are running or can run.
Mixed	**M**	*Process only*: some threads in the process are running and some are not running. Or the process is expecting some of its threads to stop.
Error <*reason*>	**E**	*Thread*: is stopped because of error *reason*. *Process*: one or more threads are in the **Error** state.

Table 6. Summary of Attached Process and Thread States (Continued)

State Name	State Character	Meaning for a thread and process
At Breakpoint	**B**	*Thread*: stopped at a breakpoint. *Process*: one or more threads are stopped at a breakpoint, but none are in the **Error** state.
Stopped <*reason*>	**T**	*Thread*: stopped because of *reason*, but not at a breakpoint and not because of an error. *Process*: one or more threads are stopped, but none are in the **At Breakpoint** state and none are in the **Error** state.
In Kernel	**K**	*Thread only:* the thread is executing inside the kernel (that is, made a system call). When a thread is in the kernel the operating system does not allow the debugger to view the full state of the thread.

The **Error** state usually indicates that your program received a fatal signal from the operating system. Some signals, such as SIGSEGV, SIGBUS, and SIGFPE may indicate an error in your program. You can control how TotalView handles signals your program receives.

Handling Signals

If your program contains a signal handler routine, you might need to adjust the way the debugger handles signals. You can change the way in which TotalView handles signals by using a dialog box (described in this section), an X resource (see “totalview*signalHandlingMode: action_list” on page 278), or a command-line option to the **totalview** command (refer to “TotalView Command Syntax” on page 287).

By default, TotalView handles UNIX signals as outlined in Table 7.

Table 7. Default Signal Handling Behavior

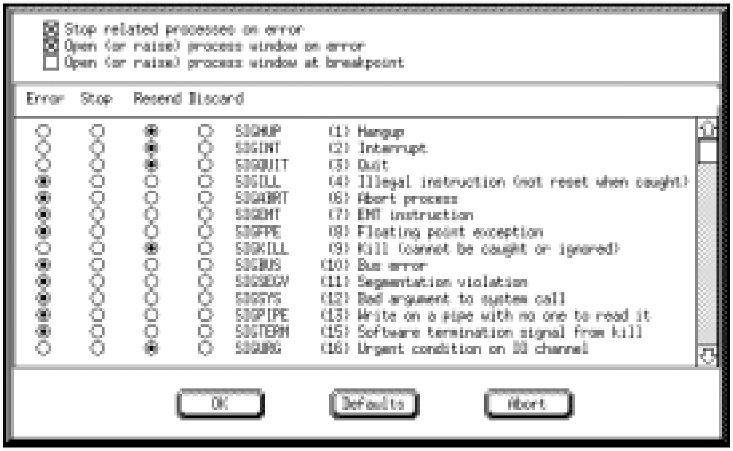
Signals that are Passed Back to Your Program	**Signals that Stop Your Program or Cause an Error**
SIGHUP	SIGILL
SIGINT	SIGTRAP
SIGQUIT	SIGIOT
SIGKILL	SIGEMT
SIGALRM	SIGFPE
SIGURG	SIGBUS
SIGCONT	SIGSEGV
SIGCHLD	SIGSYS
SIGIO	SIGPIPE
SIGVTALRM	SIGTERM
SIGPROF	SIGTSTP
SIGWINCH	SIGTTIN
SIGLOST	SIGTTOU
SIGUSR1	SIGXCPU
SIGUSR2	SIGXFSZ

Note: The SIGTRAP and SIGSTOP signals are used internally by the TotalView debugger. If the process encounters any of these signals, TotalView neither stops the process with an error nor passes the signal back to your program. Further, you cannot alter the way the debugger uses these signals.

Some hardware registers can affect how signals are handled on your platform, such as the SIGFPE signal and others. For more information, refer to “Interpreting Status and Control Registers” on page 124 and Appendix C, “Architectures,” on page 333.

If the TotalView debugger's defaults are not satisfactory, you can change the signal handling mode. To do so, go to the process window and complete the following steps:

1. Display the **Arguments/Create/Signal** submenu and select the **Set Signal Handling Mode...** command. A dialog box appears, as shown in Figure 19.



☒ Stop related processes on error
☒ Open (or raise) process window on error
□ Open (or raise) process window at breakpoint

Error	Stop	Resend	Discard		
○	○	●	○	SIGHUP	(1) Hangup
○	○	●	○	SIGINT	(2) Interrupt
○	○	●	○	SIGQUIT	(3) Quit
●	○	○	○	SIGILL	(4) Illegal instruction (not reset when caught)
●	○	○	○	SIGABRT	(6) Abort process
●	○	○	○	SIGEMT	(7) EMT instruction
●	○	○	○	SIGFPE	(8) Floating point exception
○	○	●	○	SIGKILL	(9) Kill (cannot be caught or ignored)
●	○	○	○	SIGBUS	(10) Bus error
●	○	○	○	SIGSEGV	(11) Segmentation violation
●	○	○	○	SIGSYS	(12) Bad argument to system call
●	○	○	○	SIGPIPE	(13) Write on a pipe with no one to read it
●	○	○	○	SIGTERM	(15) Software termination signal from kill
○	○	●	○	SIGURG	(16) Urgent condition on IO channel

Figure 19. Dialog Box for Set Signal Handling Mode Command

Note: The set of signal names and numbers shown in the dialog box are platform-specific. The dialog box displayed on your platform may have additional signals and different signal numbers.

2. By default, when your program encounters an error signal, TotalView stops all related processes. If you do not want this behavior, deselect the **Stop related processes on error** checkbox.
3. By default, when your program encounters an error signal, TotalView opens or raises the process window. If you do not want this behavior, deselect the **Open (or raise) process window on error** checkbox. You can change the default setting of this checkbox using an X resource ("totalview*popOnError: {on | off}" on page 275) or a command line option.

Note: If the processes in a multiprocess program encounter an error, the debugger automatically opens a process window for only the *first* process that encounters an error. Thus, if your program has many processes, this feature prevents the screen from filling up with process windows.

4. If you select the **Open (or raise) process window at breakpoint** checkbox, TotalView will open or raise the process window when your program encounters a breakpoint. If you want this behavior by default, you can change the default setting of this checkbox using an X resource (“totalview*popAtBreakpoint: {on | off}” on page 275) or a command line option

5. Scroll the signal list to the desired signal.

6. For each signal listed in the dialog box, choose one of the following signal handing modes by selecting its radio button:

   Error Stops the process, places it in the error state, and displays an error in the title bar of the process window. If the **Stop related processes on error** checkbox is selected, the debugger also stops all related processes. You should select this signal handling mode for severe error conditions, such as SIGSEGV and SIGBUS signals.

   Stop Stops the process and places it in the stopped state. Select this signal handling mode if you want the signal to be handled like the SIGSTOP signal.

   Resend Sends the signal to the process. If your program contains a signal handling routine, you should use this mode for all the signals that it handles. By default, the common signals for terminating a process (SIGKILL and SIGHUP) use this mode.

   Discard Discards the signal and restarts the process without a signal.

Note: Don’t use Discard mode for fatal signals, such as SIGSEGV and SIGBUS. If you do, the debugger can get caught in a signal/resignal loop with your program, with the signal immediately recurring because of repeated reexecution of the failing instruction.

7. Select **OK** to confirm your changes, **Abort** to cancel the changes, or **Defaults** to return to the default mode settings.

Setting Search Paths

If your source code, executable or object files reside in a number of different directories, you can set search paths in the debugger for these directories with the **Set Search Directory** command. By default, the debugger searches the following directories (in order) for source code:

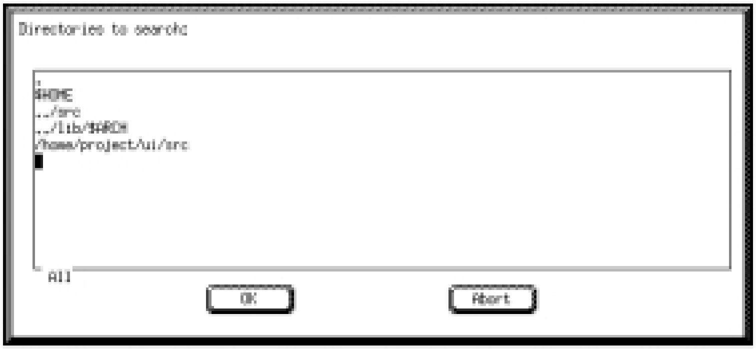
1. The current working directory (**.**).
2. The directories you specify with the **Set Search Directory** command, in the exact order you enter them in the dialog box.
3. If you specified a full pathname for the executable when you started TotalView, it searches the directory specified.
4. The directories specified in your PATH environment variable.

These search paths apply to *all* processes that you are debugging, and to *all* directory search situations in TotalView.

To use the **Set Search Directory** command, go to the process window and complete these steps:

1. Display the **Display/Directory/Edit** submenu and select the **Set Search Directory... (d)** command.

   A dialog box appears, as shown in Figure 20.



Directories to search:

```
.
$HOME
../src
../lib/$ARCH
/home/project/ui/src
```

Figure 20. Dialog Box for Set Search Directory Command

2. Enter the directories in the order you want them searched, separating each directory with a space. You can use multiple lines if needed.

   The current working directory (**.**) is the first directory listed in the window. You can move the current working directory further down the list, but if you remove it, TotalView inserts it at the top of the list again.

   You can specify relative pathnames, which are interpreted with respect to the current working directory.

3. Select **OK** (or press Shift-Return).

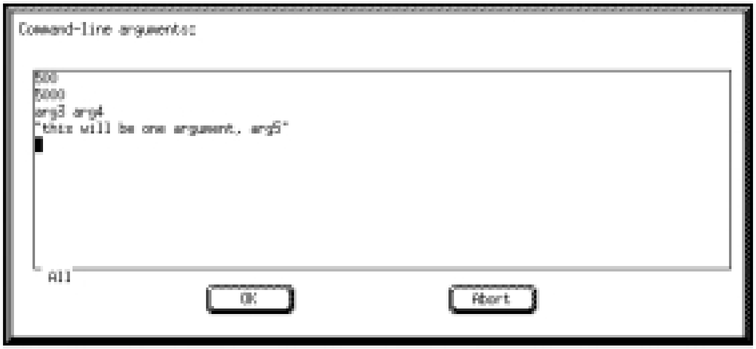
Once you change the list of directories with the **Set Search Directory** command, the debugger automatically searches again for the source file that is currently displayed in the process window.

Note: You can specify search directories that apply across TotalView sessions with an X Window System resource. Refer to “totalview*searchPath: dir1[,dir2,...]” on page 277.

Setting Command Arguments

When the debugger creates a process, it passes one argument to the program by default: the name of the file containing the executable code for the process. If your program requires any arguments from the command line, you must set these arguments *before* you start the process. To do so, go to the process window and complete the following steps:

1. Display the **Arguments/Create/Signal** submenu and select the **Set Command Arguments... (a)** command. A dialog box appears, as shown in Figure 21.



Command-line arguments:

```
500
5000
arg3 arg4
"this will be one argument, arg5"
```

Figure 21. Dialog Box for Set Command Arguments Command

2. Enter the arguments to be passed to the program. Separate each argument with a space, or place each argument on a separate line. If an argument has spaces in it, enclose the whole argument in double quotes.
3. Select **OK** (or press Shift-Return).

You can also set command-line arguments with the **–a** option of the **totalview** command, as discussed in “Starting the TotalView Debugger” on page 37.

Specifying Environment Variables

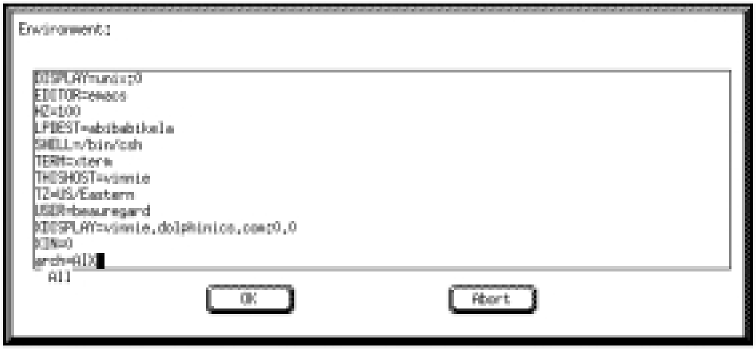
You can set and edit the environment variables that TotalView passes to a process when it creates the process. When TotalView creates a new process, it passes a list of environment variables to the process. By default, a new process inherits TotalView's environment variables, and a remote process inherits **tvdsvr**'s environment variables.

If the environment variable dialog is empty, the process inherits its environment variables from TotalView or **tvdsvr**. If you add environment variables to the dialog, the process no longer inherits its environment variables from TotalView or **tvdsvr**, it only receives the variables specified in the dialog box. Therefore, if you want to add to the variables inherited from TotalView or **tvdsvr**, you must enter all of the variables inherited into the dialog and then make your additions in the dialog.

An environment variable is specified by: *name*=*value*. For example, **DISPLAY=unix:0.0** specifies an environment variable named **DISPLAY** with the value **unix:0.0**.

To add, delete, or modify the environment variables, go to the process window and complete the following steps:

1. Display the **Arguments/Create/Signal** submenu and select the **Set Environment Variables** command. In the dialog box, you must place each environment variable on a separate line. TotalView ignores blank lines. Figure 22 shows the dialog box.



```
DISPLAY=unix:0
EDITOR=emacs
HZ=100
LPDEST=abibabikala
SHELL=/bin/csh
TERM=xterm
THISHOST=vinnie
TZ=US/Eastern
USER=beauregard
XDISPLAY=vinnie.dolphinics.com:0.0
XIN=0
arch=AIX
  AIX
```

Figure 22. Environment Variables Dialog Box

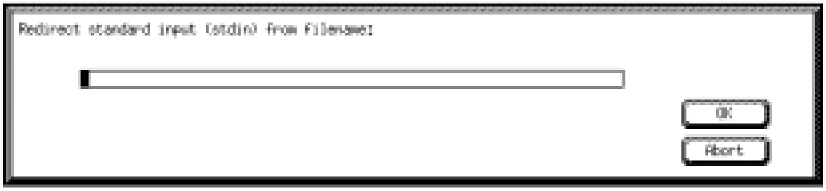
2. In the dialog box, you must place each environment variable on a separate line. TotalView ignores blank lines.
3. To change the name or value of an environment variable, edit the line.
4. To add a new environment variable, insert a new line and specify the name and value.
5. To delete an environment variable, delete the line. Deleting all the lines causes the process to inherit TotalView's or **tvdsvr**'s environment.
6. Select **OK** (or press Shift-Return).

Setting Input and Output Files

Before beginning execution of the program you're debugging, TotalView determines how to handle standard input (**stdin**) and standard output (**stdout**). By default, TotalView creates the program so that it reads **stdin** from and writes **stdout** to the shell window from which you started TotalView.

If desired, you can redirect **stdin** or **stdout** to a file. To do so, complete these steps from the process window before you start executing your program:

1. Display the **Arguments/Create/Signal** submenu and select either **Input from File... (<)** or **Output to File... (>)**. A dialog box appears. Figure 23 shows the dialog for **Input from File**.



Redirect standard input (stdin) from Filename:

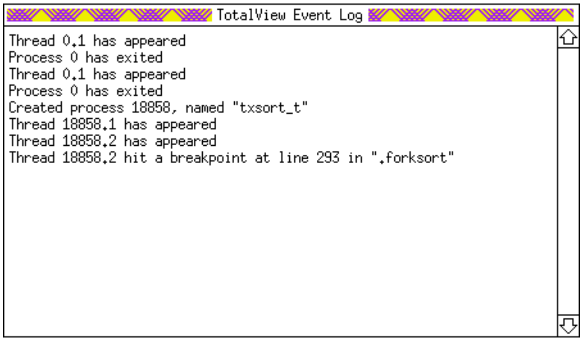
Figure 23. Dialog Box for Input from File Command

2. Enter the name of the file, relative to your current working directory.
3. Select **OK** (or press Shift-Return).

Monitoring TotalView Sessions

The TotalView debugger logs all significant events occurring for all processes you are debugging. To view the event log, go to the root window and select the **Show Event Log Window** command. The event log window displays a sequential list of events that you can scroll.

Figure 24 shows the event log window.



```
Thread 0.1 has appeared
Process 0 has exited
Thread 0.1 has appeared
Process 0 has exited
Created process 18858, named "txsort_t"
Thread 18858.1 has appeared
Thread 18858.2 has appeared
Thread 18858.2 hit a breakpoint at line 293 in ".forksort"
```

Figure 24. Event Log Window

CHAPTER 4: Setting Up Remote Debugging Sessions

This chapter explains how to set up TotalView remote debugging sessions for debugging over the network or over a serial line.

For information on how to set up a basic debugging session, see Chapter 3, “Setting Up a Debugging Session,” on page 35. For information on how to set up a parallel debugging session, see Chapter 5, “Setting Up Parallel Debugging Sessions,” on page 75.

In this chapter, you will learn how to:

- Debug remote processes
- Connecting to remote machines
- Start the debugger server for remote debugging
- Debug over a serial line

Debugging Remote Processes

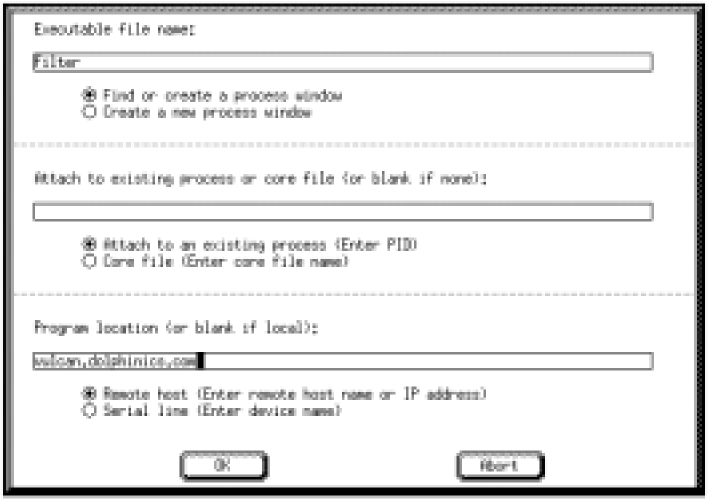
You can begin debugging remote processes either by loading a remote executable, or by attaching to a remote process.

Note: You cannot examine core files on remote nodes.

Loading a Remote Executable

To load a remote program into TotalView, do the following:

1. Complete steps in “Loading a New Executable” on page 38.
2. Enter the host name or TCP/IP address of the machine on which the executable should be running in the bottom section of the dialog box, as shown in Figure 25.



Executable file name:

filter

● Find or create a process window
○ Create a new process window

Attach to existing process or core file (or blank if none):

● Attach to an existing process (Enter PID)
○ Core file (Enter core file name)

Program location (or blank if local):

vulcan.dolphinics.com

● Remote host (Enter remote host name or IP address)
○ Serial line (Enter device name)

Figure 25. New Program Window Dialog Box

Note: On some multiprocessor platforms, there will be additional radio buttons in the lower section of the dialog box. You can use these buttons for debugging programs that are running on groups or clusters of processors.

3. Press Return.

Note: If this method does not work, you might need to disable the auto-launch feature for this connection and start the debugger server manually. In step 2, as an alternative, you can specify *hostname***:***portnumber*, where *portnumber* is the TCP/IP port number on which the debugger server (**tvdsvr**) is communicating with TotalView. For more information on this alternative, refer to “Starting the Debugger Server for Remote Debugging” on page 64.

Attaching to a Remote Process

You can attach to a remote process using the same dialog boxes as you do when you attach to a local process, but you enter information in different boxes. You can also attach to a remote process by bringing up the correct windows, then diving into processes from them.

To attach to a remote process, complete the following steps:

1. Complete the steps in “Attaching Using New Program Window” on page 42.
2. Enter the host name or TCP/IP address of the machine on which the executable should be running in the bottom section of the dialog box.

Note: On some multiprocessor platforms, there will be additional radio buttons in the lower section of the dialog box. You can use these buttons for debugging programs that are running on groups or clusters of processors.

3. Press Return.

Note: If this method does not work, you might need to disable the auto-launch feature for this connection and start the debugger server manually. In step 2, as an alternative, you can specify *hostname***:***portnumber*, where *portnumber* is the TCP/IP port number on which the debugger server (**tvdsvr**) is communicating with TotalView. For more information on this alternative, refer to “Starting the Debugger Server for Remote Debugging” on page 64.

You can also attach to a remote process by first connecting to a remote host with the **New Program Window (n)** command and then bringing up a list of unattached processes with the **Show All Unattached Processes (N)** command. You can attach to these processes by diving into them.

1. Connect to the remote host. For details on how to do this, see “Connecting to Remote Machines” on page 63.
2. After you connect to the remote host, bring up a list of unattached processes. You can attach to these processes by diving into them. For details on these steps, see “Attaching Using Show All Unattached Processes” on page 40.

Connecting to Remote Machines

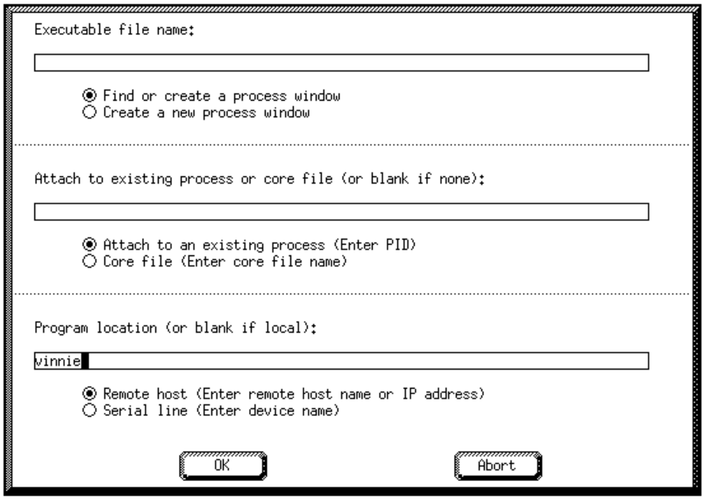
If the you want to connect to a remote machine, you can do it in two ways—by using the **–remote** option on the command line when you start TotalView or by using the **New Program Window (n)** command from the root window after you start TotalView.

If TotalView supports the runtime library (e.g., MPI, PVM, or HPF) then it automatically connects to remote hosts for you as part of the automatic process acquisition. Therefore, you do not need to manually connect to the remote machines. For more information, see Chapter 5, “Setting Up Parallel Debugging Sessions,” on page 75.

For details on the syntax for the command-line **–remote** option, see “Starting the TotalView Debugger” on page 37.

To connect to a remote host from a TotalView session, follow these steps:

1. Issue the **New Program Window (n)** command from the root window. A dialog box appears, as shown in Figure 26.



Executable file name:

● Find or create a process window
○ Create a new process window

Attach to existing process or core file (or blank if none):

● Attach to an existing process (Enter PID)
○ Core file (Enter core file name)

Program location (or blank if local):

vinnie

● Remote host (Enter remote host name or IP address)
○ Serial line (Enter device name)

Figure 26. Remote Host Connection

2. Delete the text from the **Executable file name** and **Attach to existing process or core file** fields.
3. Enter the host name or TCP/IP address of the machine on which the executable should be running in the bottom section of the dialog box.

Note: On some multiprocessor platforms, there will be additional radio buttons in the lower section of the dialog box. You can use these buttons for debugging programs that are running on groups or clusters of processors.

4. Press Return.

Starting the Debugger Server for Remote Debugging

Debugging a remote process with TotalView is identical to debugging a native process except for the following:

- The performance of your session depends on the performance of the network between the native and remote machines. If the network is overloaded, debugging can be slow. In general, we designed remote debugging to work with the speeds encountered on a LAN.
- TotalView works with another process running on the remote machine, called the TotalView Debugger Server (**tvdsvr**), to debug the remote process.

The rest of this section discusses the different ways you can start the TotalView debugger server

The Auto-Launch Feature

By default, TotalView automatically launches **tvdsvr** for you, which is known as the auto-launch feature. The advantage of auto-launch is that it makes it easy to start debugging remote processes—you don’t need to take any action to start the debugger server.

If you want to know more about auto-launch, here is the sequence of actions carried out by you, TotalView, and **tvdsvr** when auto-launch is enabled:

1. With the **New Program Window** command, you specify the host name of the machine on which you want to debug a remote process, as described in "Debugging Remote Processes" on page 60.
2. TotalView begins listening for incoming connections.
3. TotalView launches the **tvdsvr** process with the server launch command. "The Server Launch Command" on page 66 describes the command in detail.
4. The **tvdsvr** process starts on the remote machine.
5. The **tvdsvr** process establishes a connection with TotalView.

Figure 27 summarizes the actions carried out by the auto-launch feature.

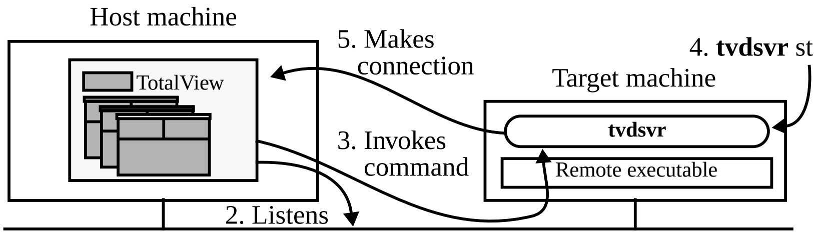


Figure 27. Auto-Launch Feature

Auto-Launch Options

If the auto-launch feature does not work on your system, you can tailor the following items:

- The command used by TotalView to launch **tvdsvr**
- The arguments passed to the launch command or to **tvdsvr**
- The length of time TotalView waits (that is, the timeout) to receive a connection from **tvdsvr**
- Whether or not the auto-launch feature is enabled

The only constraint in tailoring auto-launch is that **tvdsvr** must be started on the remote machine with the **–callback** and **–set_pw** arguments.

The Server Launch Command

By default, TotalView uses the following command string when it automatically launches the debugger server:

```
rsh %R –n "cd %D && tvdsvr –callback %L –set_pw %P –verbosity %V"
```

With this command string, the **rsh** command invokes a shell on the host name specified by **%R** and invokes the commands enclosed in quotation marks, where:

%R Expands to the host name of the remote machine that you specified in the **New Program Window** command.

–n Causes the remote shell to read standard input from **/dev/null**.

When the remote shell is started by **rsh**, it first changes to the **%D** directory with the **cd** command:

%D Expands to the full pathname of the directory to which TotalView is connected.

Note that the "**cd %D**" portion of the command assumes that the host machine and the target machine mount identical filesystems. That is, the pathname of the directory to which TotalView is connected must be identical on both the host and target machines.

Next, the remote shell starts the TotalView Debugger Server with the **tvdsvr** command and the following arguments:

–callback Establishes a connection from **tvdsvr** to TotalView using the specified host name and port number.

%L Expands to the host name and TCP/IP port number (*hostname*:*port*) on which TotalView is listening for connections from **tvdsvr**.

–set_pw Sets a 64-bit password for security. TotalView must supply this password when **tvdsvr** establishes the connection with it.

%P Expands to the password that TotalView automatically generated.

–verbosity Sets the verbosity level of the TotalView Debugger Server.

%V Expands to the current TotalView verbosity setting.

To change the server launch command each time you start TotalView, you can set an X resource. See “totalview*serverLaunchString: command_string” on page 277 for more information.

For the complete syntax of the **tvdsvr** command, refer to “TotalView Debugger Server Command Syntax” on page 299.

Changing the rsh Command

If desired, you can substitute a different command for **rsh**, but the command must invoke the **tvdsvr** process with the arguments shown (**–callback** and **–set_pw**).

Note: If you’re not sure whether **rsh** works at your site, try the "**rsh** *hostname*" command from an **xterm**, where *hostname* is the name of the host on which you want to invoke the remote process. If this command prompts you for a password, you must add the host name of the host machine to your **.rhosts** file on the target machine for TotalView to invoke **tvdsvr** properly.

For example, although the **rsh** command provides reasonable security, your site may prefer to invoke remote processes with a more secure command. As another example, you could even use a combination of the **echo** and **telnet** commands:

```
echo %D %L %P %V; telnet %R
```

Once **telnet** establishes the connection to the remote host, you could use the **cd** and **tvdsvr** commands directly, using the values of **%D**, **%L**, **%P,** and **%V** that were displayed by the **echo** command:

```
% cd directory

% tvdsvr –callback hostname:portnumber –set_pw password
```

If you have no command for invoking a remote process, you cannot use the auto-launch feature and should disable it.

For information on the **rsh** command, refer to the manual page supplied with your operating system.

Changing the Arguments

You can also change the command-line arguments passed to **rsh** (or whatever command you select to invoke the remote process).

For example, if the host machine does not mount the same filesystems as your target machine, it may need to use a different path to access the executable to be debugged. If this is the case, you could change **%D** to an appropriate directory on the target machine.

If your remote executable reads from standard input, you cannot use the **–n** option with **rsh** because this causes the remote executable to receive an EOF immediately on standard input. If you omit **–n**, the remote executable reads standard input from the **xterm** in which you started TotalView. Therefore, if your remote program reads from standard input, you should invoke **tvdsvr** from an **xterm** window. Use the following command string to launch the debugger server:

```
rsh %R –n "cd %D && xterm –display hostname:0 –e tvdsvr –callback %L –set_pw %P
   –verbosity %V"
```

Now, each time TotalView launches **tvdsvr**, a new **xterm** appears on your screen to handle standard input and output for the remote program.

The Connection Timeout

When TotalView automatically launches **tvdsvr**, it waits for 30 seconds to receive a successful connection from **tvdsvr**. If TotalView receives nothing, it times out. If desired, you can specify a timeout of anywhere between 1 and 3600 seconds (1 hour).

Note: If you notice that TotalView fails to launch **tvdsvr** (as shown in the **xterm** window from which you started TotalView) before the timeout expires, you can press CTRL-C in any TotalView window to have TotalView terminate the launch. Otherwise, TotalView terminates the launch when the timeout occurs.

To change the timeout for every TotalView session, you can set an X resource. See “totalview*serverLaunchTimeout: n” on page 278 for more information.

Disabling Auto-Launch

If changing the auto-launch options will not make the auto-launch feature useful for you, you can disable the auto-launch feature and start **tvdsvr** manually. You can disable the auto-launch feature in several different ways:

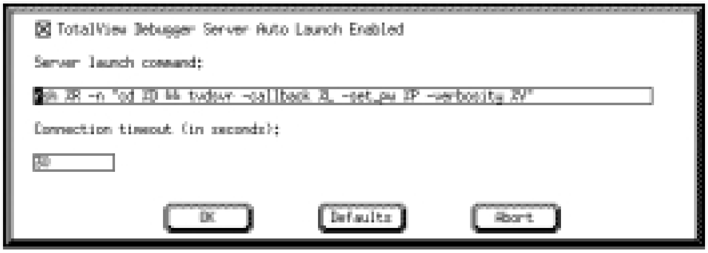
- When you change the auto-launch options, as described in “Changing the Options” on page 69, deselect the **TotalView Debugger Server Auto Launch Enabled** checkbox at the top of the dialog box. This disables auto-launch for your current TotalView session.
- When you debug the remote process, as described in “Debugging Remote Processes” on page 60, specify both a host name and port number in the bottom section of the **New Program Window** dialog box. This disables auto-launch for the current connection.
- Set an X resource that disables auto-launch, as described in “totalview*serverLaunchEnabled: {true | false}” on page 277. This disables auto-launch for every TotalView session.

Note: If you disable the auto-launch feature, you must start **tvdsvr** *before* you load a remote executable or attach to a remote process.

Changing the Options

To actually change the server launch command or the connection timeout used by TotalView to launch **tvdsvr**, or to actually disable the auto-launch feature entirely, you use the server launch window command. To do so:

1. From the root window, select the **Server Launch Window** menu command. A dialog box appears, as shown in Figure 28.



☒ TotalView Debugger Server Auto Launch Enabled

Server launch command:

rsh %R -n "cd %D && tvdsvr -callback %L -set_pw %P -verbosity %V"

Connection timeout (in seconds):

30

Figure 28. Dialog Box for Launching Debugger Server

2. Change the desired options.
3. Press Return.

Note: If you make a mistake or decide you want to revert to the default option settings in the dialog, select the **Defaults** button. You can revert to the default settings even if you used an X resource to change the settings. Then, to apply the original option settings, you need to select the **OK** button.

Starting the Debugger Server Manually

If you cannot tailor the auto-launch feature to work on your system, you can start the debugger server manually if needed. The disadvantage of this method is that it is insecure: other users could connect to your instance of **tvdsvr** and begin using your UNIX UID.

To start **tvdsvr** manually:

1. From the root window, select the **Server Launch Window** command. A dialog box appears, as shown in Figure 28.
2. Deselect the **TotalView Debugger Server Auto Launch Enabled** checkbox to disable the auto-launch feature.
3. Press Return.
4. Log in to the remote machine and start **tvdsvr**:

   ```
   % tvdsvr –server
   ```

   The **tvdsvr** command prints out the port number used and the password assigned and then begins listening for connections. Be sure to make note of the password; you’ll need to enter it later in step 9.

   If the default port number (4142) is not suitable, you need to use the **–port** or **–search_port** options with the **tvdsvr** command. For details, refer to “TotalView Debugger Server Command Syntax” on page 299.

5. From the root window in TotalView, select the **New Program Window** command. A dialog box appears.
6. Enter the name of the executable in the top of the dialog.
7. Enter the *hostname*:*portnumber* in the bottom of the dialog.

8. Press Return.

   TotalView now attempts to establish a connection to **tvdsvr**.

9. When TotalView prompts you for the password, enter the password that **tvdsvr** displayed in step 4.

Figure 29 summarizes the steps used when you start **tvdsvr** manually.

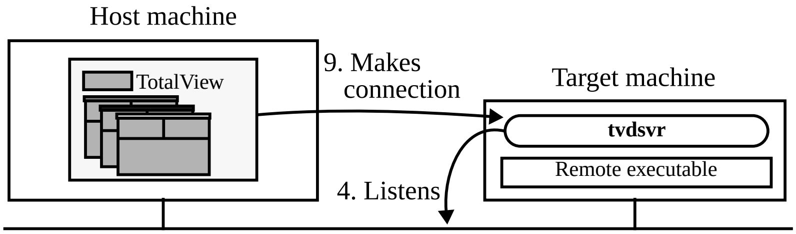


Figure 29. Manual Launching of Debugger Server

Debugging Over a Serial Line

In addition to debugging over a TCP/IP socket connection, TotalView allows you to debug over a serial line. However, in cases where a network connection exists, you will probably want to use TCP/IP sockets remote debugging for better performance.

You will need to have two connections to the target machine. One connection will be for the console and the other dedicated for use by TotalView. Do not try to use one serial line; TotalView cannot share a serial line with the console.

Figure 30 shows an example TotalView debugging session over a serial line. In this example, TotalView is running on a host machine and communicating over a dedicated serial line with the TotalView Debugger Server running on the target host. A VT100 terminal is connected to the target host's console line which allows you to type commands on the target host.

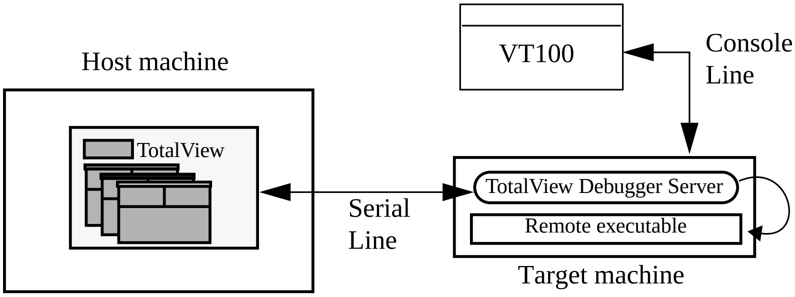


Figure 30. TotalView Debugging Session over a Serial Line

Start the TotalView Debugger Server

To start a TotalView debugging session over a serial line from the command line, you must first start the TotalView debugger Server.

Through the console connected to the target machine, issue the command to start the TotalView Debugger Server (**tvdsvr**) and specify the name of the serial port device on the target machine. The syntax of the TotalView Debugger Server command is:

```
% tvdsvr –serial device[:options]
```

where *device* is the name of the serial line device and *options* are options to control the serial line on the target machine. The TotalView Debugger Server will wait for TotalView to establish a connection.

For example:

```
% tvdsvr –serial /dev/com1:baud=38400
TotalView Debugger Server 3.8.1 (ICCDP protocol level 17, rev 15)
Copyright 1996-1998 by Dolphin Interconnect Solutions, Inc.  ALL
RIGHTS RESERVED.
Copyright 1989-1996 by BBN Inc.
```

Currently the only option you are allowed to specify is the baud rate, which defaults to **38400**.

Starting TotalView on a Serial Line

Start TotalView on the host machine and include the name of the serial line device. The syntax of the TotalView command is:

```
% totalview –serial device[:options] filename
```

where *device* is the name of the serial line device on the host machine, *options* are options to control the serial line on the host machine and *filename* is the name of the executable file. TotalView will connect to the TotalView Debugger Server.

For example:

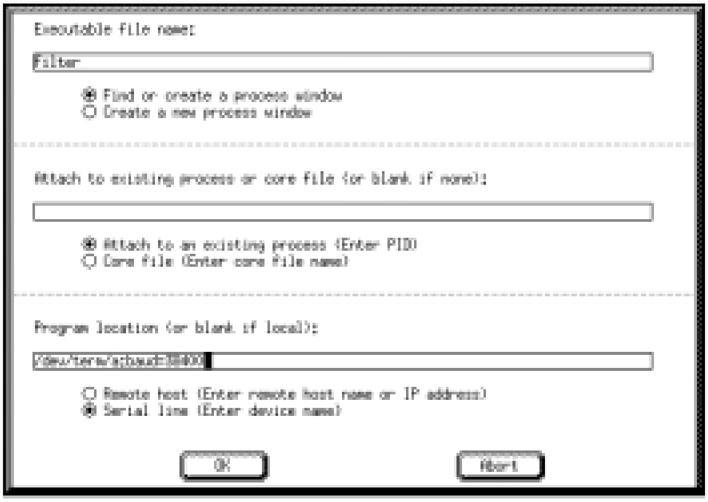
```
% totalview –serial /dev/term/a test_pthreads
```

Currently the only option you are allowed to specify is the baud rate, which defaults to **38400**.

New Program Window

To start a TotalView debugging session over a serial line when you are already in TotalView, do the following:

1. Start the TotalView Debugger Server. See “Start the TotalView Debugger Server” on page 73.
2. Issue the **New Program Window (n)** command from the root window to display the **New Program Window** dialog box, shown in Figure 31.
3. Enter the name of the executable file in the **Executable file name** field.
4. Enter the name of the serial line device in the **Program location** field, and select the **Serial line** radio button.
5. Press Return or select OK.



Executable file name:

Filter

● Find or create a process window
○ Create a new process window

Attach to existing process or core file (or blank if none):

● Attach to an existing process (Enter PID)
○ Core file (Enter core file name)

Program location (or blank if local):

/dev/term/a;baud:38400

○ Remote host (Enter remote host name or IP address)
● Serial line (Enter device name)

Figure 31. New Program Window Dialog Box

CHAPTER 5: Setting Up Parallel Debugging Sessions

This chapter explains how to set up TotalView parallel debugging sessions for MPI, PVM, or Portland Group HPF applications. In this chapter, you will learn how to debug:

- MPI and IBM PE applications
- PVM or DPVM applications
- Portland Group HPF applications

For tips on debugging parallel applications, see “Parallel Debugging Tips,” on page 110.

For information on how to set up a basic debugging session, see Chapter 3, “Setting Up a Debugging Session,” on page 35.

For information on how to set up a remote debugging session and on the TotalView debugger server, see Chapter 4, “Setting Up Remote Debugging Sessions,” on page 59.

Debugging MPI Applications

You can use TotalView to debug your Message Passing Interface (MPI) programs. With TotalView, you can:

- Automatically acquire processes at start-up
- Attach to a parallel program and automatically acquire the parallel processes
- Display the message queue state of a process

Automatic process acquisition at start-up is supported for the MPI implementations:

- MPICH version 1.1.0 or later running on any platform that is supported by both TotalView and MPICH (see "Debugging MPICH Applications," on page 77)
- Digital MPI (DMPI) running on Digital Unix on Alpha (see "Debugging Digital MPI Applications," on page 81)
- IBM MPI Parallel Environment (PE) running on AIX on RS/6000 and SP (see "Debugging IBM MPI (PE) Applications," on page 82)
- SGI MPI running on IRIX on MIPS processors (see "Debugging SGI MPI Applications," on page 86)

For more information on message queue display, see "Displaying Message Queue State," on page 87.

For tips on debugging parallel applications, see "Parallel Debugging Tips," on page 110.

Debugging MPICH Applications

To debug Message Passing Interface/Chameleon Standard (MPICH) applications you must use MPICH version 1.1.0 or later on a homogenous collection of machines. If you need a copy of MPICH, it is available at no cost from Argonne National Laboratory at **http://www.mcs.anl.gov/mpi**.

Note: Please see the TotalView release notes for information on how to patch your MPICH 1.1.0 distribution.

You should configure the MPICH library to use either the **ch_p4**, **ch_shmem**, **ch_lfshmem**, or **ch_mpl** devices. For networks of workstations, **ch_p4** is the normal default. For shared-memory SMP machines, **ch_shmem** is the default. On an IBM SP machine, use the **ch_mpl** device. The MPICH source distribution includes all of these devices and you can choose which to use when you configure and build MPICH on your machine.

Note: When you configure MPICH, you must ensure that the MPICH library maintains all of the information required by TotalView. Use the **–debug** option with the MPICH **configure** command.

See “Displaying Message Queue State,” on page 87 for message queue display.

Starting TotalView on an MPICH Job

You must have both TotalView (**totalview**) and the TotalView Debugger Server (**tvdsvr**) in your path when you start an MPICH job under TotalView’s control. Use the MPICH **mpirun** command that you customarily use and add the **–tv** flag:

```
% mpirun [MPICH-arguments] –tv program [program-arguments]
```

For example:

```
% mpirun –np 4 –tv sendrecv
```

The MPICH **mpirun** command uses the value of the environment variable **TOTALVIEW** as the command that starts the first process in the parallel job. Therefore, by setting this environment variable, you can use a different TotalView, or pass command line options to TotalView.

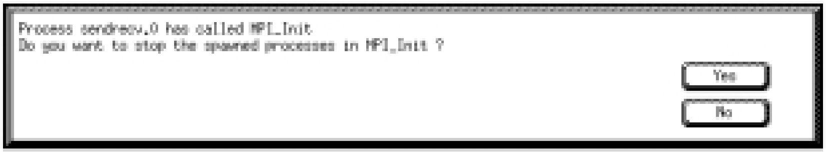
For example, you can make **mpirun** invoke TotalView with the **–no_stop_all** flag by issuing the C shell command:

```
% setenv TOTALVIEW "totalview –no_stop_all"
```

On workstations, TotalView starts the first process of your job, the master process, under the control of the debugger. Then, you can set breakpoints, and debug your code as usual.

On the IBM SP machine, the **mpirun** command uses IBM's **poe** command to start an MPI job. The MPICH **mpirun** command must still be used on the SP to start an MPICH job., including the use of the **–tv** flag. However, the details of process start-up are different since **poe** is being used to start the MPI program. For details of using TotalView with **poe**, see "Starting TotalView on a PE Job," on page 83.

When you let code run through the call to **MPI_Init()**, TotalView automatically acquires the other processes that make up your parallel job. A dialog box appears asking if you want to stop the spawned processes. This allows you to stop all of the processes in **MPI_Init()** so you can check their states before they run too far. See Figure 32.



Process sendrecv.0 has called MPI_Init
Do you want to stop the spawned processes in MPI_Init ?

Figure 32. Dialog Box for Stopping Spawned Processes

Answer **Yes**, or type **y**, if you want to stop the spawned processes.

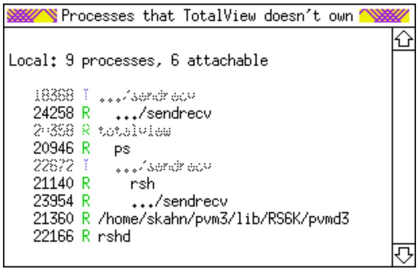
Answer **No**, or type **n**, if you want the processes to continue to run.

TotalView automatically copies breakpoints from the master process to the slave processes as it acquires them. This allows you to set up breakpoints in the slave processes by placing them in the master process. You do not have to first stop the slave processes in **MPI_Init()**. Next, TotalView updates the root window to show all the newly acquired processes.

Attaching to an MPICH Job

TotalView allows you to attach to an MPICH application even if it was not started under the control of the debugger. To attach to a running MPICH job, do the following:

1. Start TotalView in the normal manner. See “Starting the TotalView Debugger,” on page 37.
2. Issue the **Show All Unattached Processes (N)** command from the root window. A new window appears on your screen displaying the Processes that TotalView doesn’t own window, as shown in Figure 33.



```
Local: 9 processes, 6 attachable

   18368 T .../sendrecv
   24258 R   .../sendrecv
   21358 R totalview
   20946 R   ps
   22672 T   .../sendrecv
   21140 R     rsh
   23954 R     .../sendrecv
   21360 R /home/skahn/pvm3/lib/RS6K/pvmd3
   22166 R rshd
```

Figure 33. Processes that TotalView doesn’t own Window

3. On workstation clusters, attach to the first MPICH process.

   Normally, the first MPICH process is the highest process with the correct image name in the process list. Other instances of the same executable will either be

   - The **p4** listener processes if you have configured MPICH with **ch_p4**
   - Additional slave processes if you have configured MPICH with **ch_shmem** or **ch_lfshmem**
   - Additional slave processes if you have configured MPICH with **ch_p4** and have a machine file that places multiple processes on the same machine
   - On an IBM SP, attach to the **poe** process that started your job. For details, see “Starting TotalView on a PE Job,” on page 83.

   Dive into this process to attach to it.

4. After you attach to the processes, TotalView asks if you also wish to attach to the slave MPICH processes. If you do, press **Return** or choose **Yes**. If you do not, select **No**.

   If you choose **Yes**, TotalView starts the server processes and acquires all of the MPICH processes.

In some situations, the processes you expected to see may not exist (for example, they may have crashed or exited). TotalView acquires all the processes it can and then warns you if it could not attach to some of them. You can debug the processes TotalView did acquire. If you attempt to dive into a process that no longer exists (for example, through the source or target fields of a message state display), TotalView gives you a message that the requested process no longer exists.

MPICH P4 procgroup Files

If you are using MPICH with an explicit P4 procgroup file (by using the **–p4pg** flag), you must make sure you use the *same* absolute path name in your procgroup file and on the mpirun command line. If your procgroup file contains different path names that resolve to the same executable, TotalView treats each path name as a separate instance of the executable, which causes debugging problems.

You must use the *same* absolute pathname of the executable on both the TotalView command line and in the procgroup file. For example:

```
% cat  p4group
local 1 /users/smith/mympichexe
bigiron 2 /users/smith/mympichexe
% mpirun  –p4pg  p4group  –tv  /users/smith/mympichexe
```

In this example, TotalView does the following:

1. Reads the symbols from the executable **mympichexe** only once
2. Places MPICH processes in the same TotalView share group
3. Names the processes **mypichexe.0**, **mympichexe.1**, **mympichexe.2**, and **mympichexe.3**.

If Totalview assigns names such as **mympichexe<mympichexe>.0**, there is a problem and you should check the contents of your procgroup file and **mpirun** command line.

Debugging Digital MPI Applications

You can debug Digital MPI applications on the Digital UNIX Alpha platform. To use TotalView with Digital MPI, you must use Digital MPI version 1.7.

See “Displaying Message Queue State,” on page 87 for message queue display.

Starting TotalView on a Digital MPI Job

Digital MPI programs are normally started with the **dmpirun** command. To start under the control of TotalView, simply use TotalView as if you were debugging **dmpirun**.

```
% totalview dmpirun –a dmpirun-command-line
```

TotalView will start up and show you the code for the main program in **dmpirun**. Since this is not normally of interest, you should let the program run by using the **Go Process (g)** command.

The **dmpirun** command runs and starts all of the MPI processes. TotalView will acquire them and then ask you whether you want to stop them all.

Note: There may be problems with re-running Digital MPI programs under TotalView control. These have to do with resource allocation issues within Digital MPI. Consult the Digital MPI manuals and release notes for information on how to clean up the MPI system state using **mpiclean**.

Attaching to a Digital MPI Job

To attach to a running Digital MPI job, attach to the **dmpirun** process that started the job. Once you have attached to the **dmpirun** process, TotalView displays the same dialogue as it does with MPICH. (See step 4 on page 80, included in “Attaching to an MPICH Job,” on page 79.)

Debugging IBM MPI (PE) Applications

You can debug IBM MPI Parallel Environment (PE) applications on the IBM RS/6000 and SP platforms.

To take advantage of TotalView's automatic process acquisition capabilities, you must be running release 2.2 or later of the Parallel Environment for AIX. If you aren't running release 2.2, you can run TotalView on release 2.1 if you also load PTF 15.

See "Displaying Message Queue State," on page 87 for message queue display.

Preparing to Debug a PE Application

To debug a PE application, you need to prepare by doing the following:

1. If you are using switch-based communications (either "IP over the switch" or "user space") on an SP machine, you must configure your PE debugging session so that TotalView can use "IP over the switch" for communicating with the TotalView Debugger Server, by setting **adaptor_use** to **shared** and **cpu_use** to **multiple**.

   Set these up by doing at least one of the following:

   - If you are using a PE host file, add **shared multiple** after all host names or pool IDs in the host file.
   - Whether or not you have a PE host file, enter the following arguments on the **poe** command line:

     –adaptor_use shared –cpu_use multiple
   - If you do not want to set the above arguments in the **poe** command line, set the following environment variables before starting **poe**:

   ```
   % setenv MP_ADAPTOR_USE shared

   % setenv MP_CPU_USE multiple
   ```

   When using "IP over the switch," the default is usually shared adapter use and multiple cpu use, but to be safe, set it explicitly using one of the above techniques.

2. You have to be able to use remote login using **rsh**. To do this, add the host name of the remote node to the **/etc/hosts.equiv** file or to your **.rhosts** file.

   When the program is using switch-based communications, TotalView tries to start the TotalView Debugger Server using the **rsh** command with the switch host name of the node.

3. When you are using switch-based communications, you must run TotalView on one of the SP or SP2 nodes. Since TotalView uses “IP over the switch” in this case, you cannot run TotalView on an RS/6000 workstation.
4. TotalView automatically sets the **timeout** value at 600 seconds. If you get communications time-outs, you may need to set the value at a higher number, as in the following example:

   ```
   % setenv MP_TIMEOUT 1200
   ```

Note: **timeout** cannot be set through the **poe** command line.

Starting TotalView on a PE Job

Parallel Environment (PE) programs can normally be run directly from the command line with the following syntax:

```
% program [arguments] [PE_arguments]
```

They can also be run under the control of the **poe** command, as in the following:

```
% poe program [arguments] [PE_arguments]
```

However, TotalView is different in this regard. If you start TotalView on a PE application, it requires that you start on the **poe** command. The syntax of the command is:

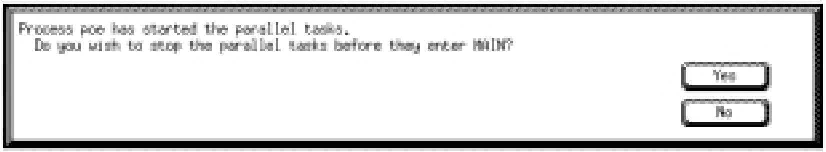
```
% totalview poe –a program [arguments] [PE_arguments]
```

For example:

```
% totalview poe –a sendrecv 500 –rmpool 1
```

Setting Breakpoints

After TotalView is running, you can start the **poe** process, which in turn, starts the parallel processes. Issue the **Go Process (g)** command from the process window. A dialog box comes up asking if you want to stop the parallel tasks. See Figure 34.



Process poe has started the parallel tasks.
Do you wish to stop the parallel tasks before they enter MAIN?

Figure 34. Parallel Tasks Dialog Box

If you want to set breakpoints in your code at this point, answer **Yes** to stop the processes. TotalView initially stops the parallel tasks, so you can set breakpoints. A program window for the first parallel task appears, in which you can set breakpoints and control the parallel tasks, using normal TotalView commands.

If you have already set and saved breakpoints in a file and you want to reload the file, may answer **No**. The parallel tasks continue running, but first TotalView automatically reloads your breakpoints.

Starting Parallel Tasks

After you set breakpoints, you can start all of the parallel tasks by issuing the **Go Group (G)** command from the parallel task program window.

Note: None of the parallel tasks will get to the first line of code in **main** until all of the parallel tasks have started.

You should be very cautious in placing breakpoints at or before the line that contains the call to **MPI_Init** (or **MPL_Init**), because time-outs occur during the initialization process. Once any of the parallel processes is allowed to proceed into the **MPI_Init** or **MPL_Init** call, all of the parallel processes should be allowed to proceed through this call within a short time. For more information on this, see “Avoiding unwanted time-outs,” on page 112.

Attaching to a PE Job

To take full advantage of TotalView's **poe**-specific automation, you need to attach to **poe** itself, and let TotalView automatically acquire the **poe** processes on its various nodes. This set of acquired processes will include the process(es) you want to debug.

You attach to the **poe** processes the same way you attach to other processes. For details on attaching to processes, see "Attaching to Processes," on page 40.

Attach from a Node Running poe

To attach TotalView to **poe** from the node running **poe**, start TotalView in the directory of the debug target. If you cannot start TotalView in the debug target directory, you can start TotalView by editing the TotalView Debugger Server (**tvdsvr**) command line before attaching to **poe**. See "The Server Launch Command," on page 66.

In the TotalView root window, bring up the unattached processes window, find the **poe** process list in your root window, and attach to it by diving into it. TotalView launches TotalView Debugger Servers as necessary.

TotalView updates the root window and opens a process window for the **poe** process, which you just dove on. In the root window, find the process you want to debug and dive on it to open a process window from which you can control and debug the target process.

If some source code is available on-line but does not display in the source code pane of the process window, you may have to issue the **Display/Directory/Edit (d)** command and specify more directories to search.

Attach from Node Not Running poe

To attach TotalView to **poe** from a node not running **poe**, follow the same procedures as in attaching from a node running **poe**, except, since you did not run TotalView from the node running **poe** (the start-up node), you will not be able to see **poe** on the process list in your root window and you will not be able to start it by diving into it.

To get **poe** on the process list in your root window, connect TotalView to the start-up node. For details on how to do this, see "Connecting to Remote Machines," on page 63 and "Attaching to Processes," on page 40. Then, update the list of processes in the Processes that TotalView doesn't own window by selecting **Update Process List (u)** from the menu. In the area headed <*startup_node_name*>, look for the process named **poe** and continue as if attaching from a node running **poe**.

Debugging SGI MPI Applications

TotalView can acquire processes started by SGI MPI version 3.1 – part of the Message Passing Toolkit (MPT) 1.2 package.

Message queue display is supported by release 1.3 of the Message Passing Toolkit. See “Displaying Message Queue State,” on page 87 for message queue display.

Starting Totalview with SGI MPI

To start an SGI MPI program under TotalView control use TotalView as if you were debugging **mpirun** itself:

```
% totalview mpirun –a mpirun-command-line
```

TotalView starts up and shows you the machine code for the SGI MPI **mpirun**. Since you are not usually interested in debugging this you should let the program run by using the **Go Process** (**g**) command.

The SGI MPI **mpirun** command executes and starts all of the MPI processes. TotalView acquires them and then asks if you want to stop them at start-up. If you do stop them, TotalView halts them before they enter the main program. You can then enter breakpoints as appropriate.

If you set a verbosity level that allows informational messages, TotalView also prints a message showing the name of the array and the value of the array services handle (ash) to which it is attaching.

Attaching to an SGI MPI Job

To attach to a running SGI MPI job, attach to the SGI MPI **mpirun** process that started the job. Once you have attached to the SGI MPI **mpirun** process, TotalView displays the dialog as it does with MPICH. (See step 4 on page 80, included “Attaching to an MPICH Job,” on page 79.)

Displaying Message Queue State

The TotalView message queue display (MQD) feature allows you to display the message queue state of your MPI program. This is a very useful debugging feature for determining the cause of message passing deadlocks.

To use the message queue display feature, you must have the correct version of MPI for your platform, as follows:

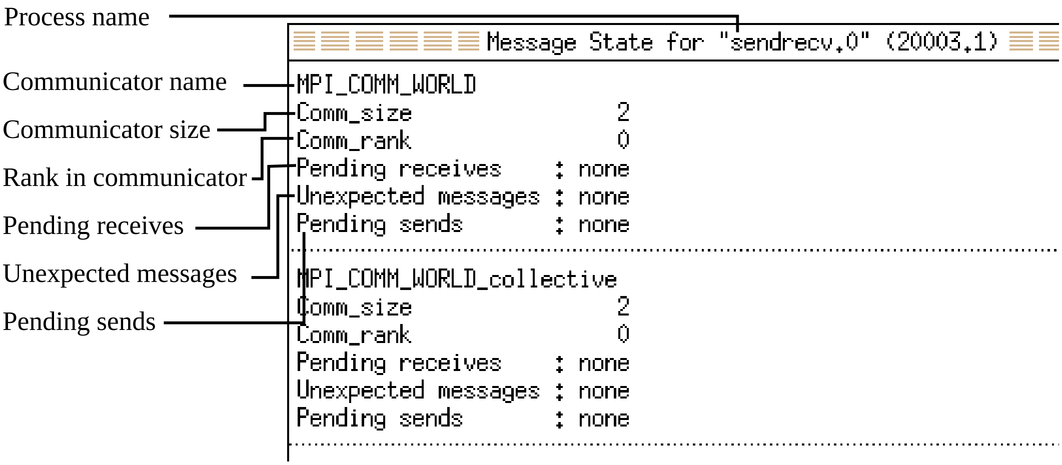
- MPICH version 1.1.0 or 1.1.1
- Digital MPI (DMPI) version 1.7
- IBM MPI Parallel Environment (PE) version 2.3 or 2.4; but *only* for programs using the *threaded* IBM MPI libraries. This functionality is not available with earlier releases, or with the non-thread-safe version of the IBM MPI library, since these libraries do not maintain information accessible to TotalView. Therefore, to use the TotalView MQD feature with IBM MPI applications, you should compile and link your code using the **mpcc_r**, **mpxlf_r**, or **mpxlf90_r** compilers.
- For SGI MPI TotalView message queue display, you must obtain the Message Passing Toolkit (MPT) release 1.3. Check with SGI for availability. TotalView contains the necessary changes to display message queue state with this version of SGI MPI, so no TotalView changes should be required.

Message Queue Display Basics

After an MPI process returns from the call to **MPI_Init()**, you can display the internal state of the MPI library by issuing the **Message State Window (m)** command in the **Process State Info** submenu of the process window. TotalView opens a message state window for the process, as shown in Figure 35.

The contents of the message state window are valid only when the process is stopped. The message state window displays the state of each of the MPI communicators that exist in the process. In some MPI implementations, such as MPICH, each user-visible communicator is implemented as two internal communicator structures, one for point-to-point, the other for collective operations. TotalView shows both structures.

Note: You cannot edit any of the fields in the message state window.



```
Message State for "sendrecv.0" (20003.1)
MPI_COMM_WORLD
Comm_size              2
Comm_rank              0
Pending receives     : none
Unexpected messages : none
Pending sends        : none
MPI_COMM_WORLD_collective
Comm_size              2
Comm_rank              0
Pending receives     : none
Unexpected messages : none
Pending sends        : none
```

Figure 35. Message State Window

For each communicator, TotalView displays the following fields:

- Name of the communicator. MPI names the pre-defined communicators such as **MPI_COMM_WORLD**. Note:
  - MPICH 1.1 and Digital Unix MPI also provide the MPI-2 communicator naming functions, **MPI_NAME_PUT** and **MPI_NAME_GET**, so you can associate a name with a communicator. If you use **MPI_NAME_PUT** to name a communicator, TotalView uses the name you gave it when displaying the communicator, so you do not have to guess which communicator is which.
  - IBM MPI and SGI MPI do not implement the MPI-2 communicator naming functions, therefore only pre-defined communicators are named. For user-created communicators, the integer value that represents the communicator is displayed. This is the value that a variable of type **MPI_Communicator** has if it represents the given communicator.
- **Comm_size** gives the number of processes in the communicator. This is the same as the result of **MPI_Comm_size()** applied to the communicator.
- **Comm_rank** gives the rank in the communicator of the process which owns the message state window. This is the same as the result of **MPI_Comm_rank()** applied to the communicator.

- List of pending receive operations.
- List of pending unexpected messages (i.e., messages that have been sent to this communicator but have not yet matched with a receive).
- List of pending send operations.

Message Operations

For each communicator, TotalView displays a list of pending receive operations, pending unexpected messages, and pending send operations. Each operation has index value displayed in square brackets (**[***n***]**), and each operation may include the following fields:

Function The MPI function (IBM MPI only). The name of the MPI function associated with the operation, e.g., **MPI_Irecv**.

Type The MPI data type (IBM MPI only). The MPI data type associated with the operation, e.g., **MPI_INT**.

Status The status of the operation. Operation status can be **Pending**, **Active**, or **Complete**.

Source or **Target** The source or target process. Source is the process from which the message should be received. Target is the process to which the message is being sent. This field shows the index of the process in the communicator, and the process name in parentheses. Dive into this field to display a process window. If the message is being received from **MPI_ANY_SOURCE**, then the display will show **ANY**.

Actual Source For receive operations, if the **Status** is **Complete** and the **Source** is **ANY**, the receiving process.

Tag The tag value. If the message is being received with **MPI_ANY_TAG**, then the display will show **ANY**.

Actual Tag For receive operations, if the **Status** is **Complete** and the **Tag** value is **ANY**, the received tag value.

User Buffer, **System Buffer**, or **Buffer**
The address of the buffer. Dive into this field to view a data window displaying the buffer contents.

Buffer Length or **Received Length**
The buffer length in bytes, shown in decimal and hexadecimal.

MPI Process Diving

To display more detail, you can dive into certain fields in the message state window. When you dive into a process field, TotalView does one of the following:

- Raises the relevant process window if it exists
- Focuses an existing process window on the requested process
- If no suitable process window exists, creates a new process window for the process

If there is no relevant process window and you want TotalView to create a new process window instead of refocusing an existing process window, hold down the Shift key with the dive button.

MPI Buffer Diving

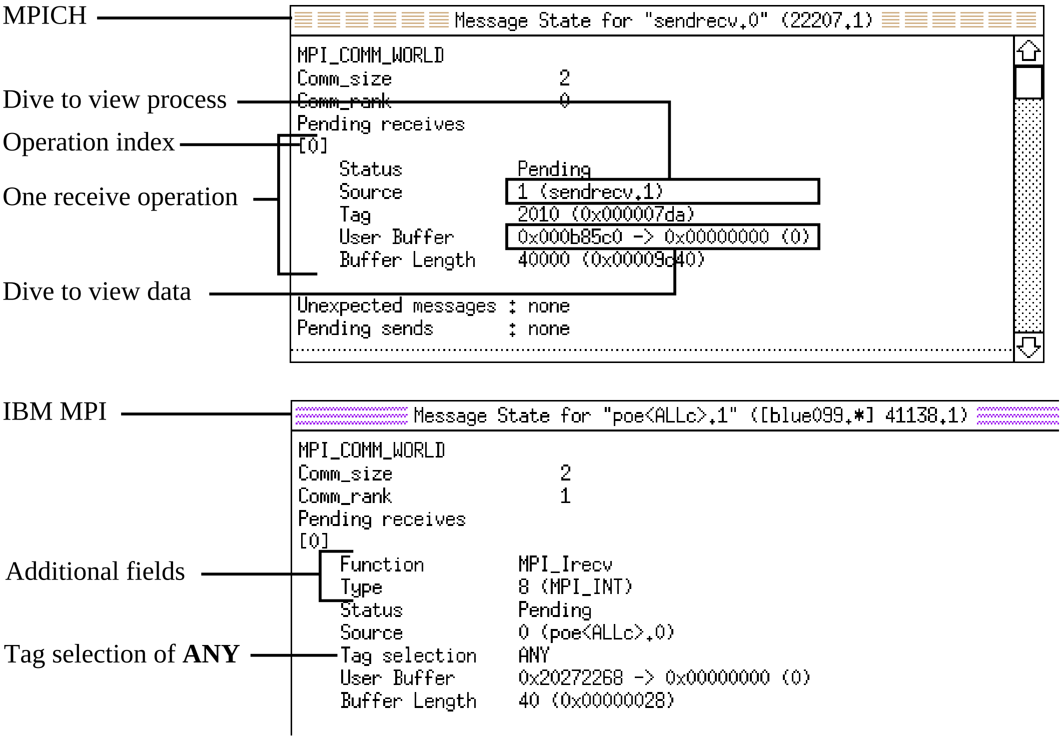
You can also dive into the buffer fields, causing a normal data window to open. TotalView attempts to guess the correct format for the data, based on the length and alignment of the buffer. If TotalView guesses incorrectly, you can edit the type field in the data window, as usual.

Note: TotalView currently does not set the buffer type using the MPI data type. Some MPI implementations, such as MPICH, do not maintain the type information. IBM MPI does maintain the data type, however TotalView does not yet use it for formatting the data buffer.

Pending Receive Operations

TotalView displays each pending receive operation in the pending receives list. Figure 36 shows examples of MPICH and IBM MPI pending receive operations.

Note: TotalView displays all of the receive operations that are maintained by the IBM MPI library. You should set the environment variable **MP_EUIDEVELOP** to the value **DEBUG** if you want blocking operations to be visible, otherwise only non-blocking operations are maintained. For more details on the **MP_EUIDEVELOP** environment variable, consult the IBM manual *Parallel Environment Operations and Use*.



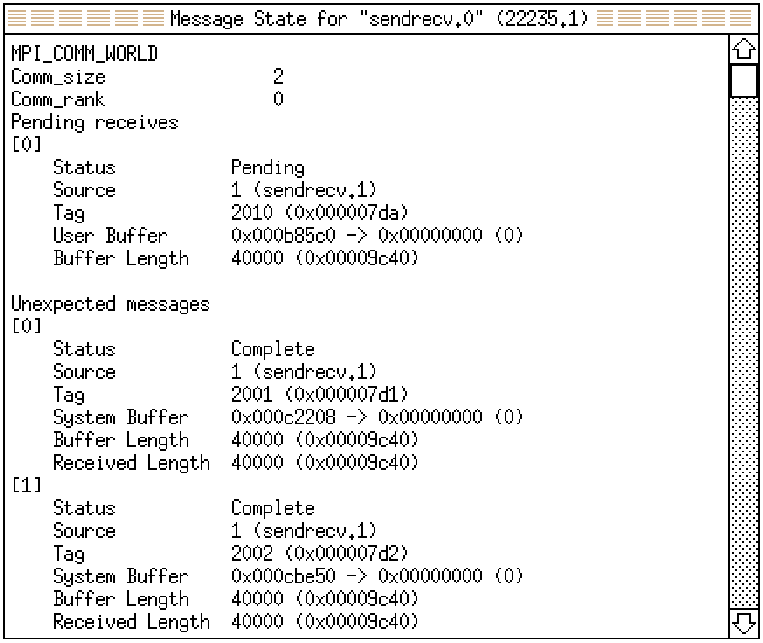
```
Message State for "sendrecv.0" (22207.1)
MPI_COMM_WORLD
Comm_size                 2
Comm_rank                 0
Pending receives
[0]
    Status                Pending
    Source                1 (sendrecv.1)
    Tag                   2010 (0x000007da)
    User Buffer           0x000b85c0 -> 0x00000000 (0)
    Buffer Length         40000 (0x00009c40)

Unexpected messages : none
Pending sends       : none
```

```
Message State for "poe<ALLc>.1" ([blue099.*] 41138.1)
MPI_COMM_WORLD
Comm_size                 2
Comm_rank                 1
Pending receives
[0]
    Function              MPI_Irecv
    Type                  8 (MPI_INT)
    Status                Pending
    Source                0 (poe<ALLc>.0)
    Tag selection         ANY
    User Buffer           0x20272268 -> 0x00000000 (0)
    Buffer Length         40 (0x00000028)
```

Figure 36. Message State Pending Receive Operation

Unexpected Messages

The unexpected messages portion of the display shows the envelope information for messages that have been sent to this communicator in this process, but which have not yet been matched by a receive operation. Figure 37 shows an example of MPICH unexpected messages.

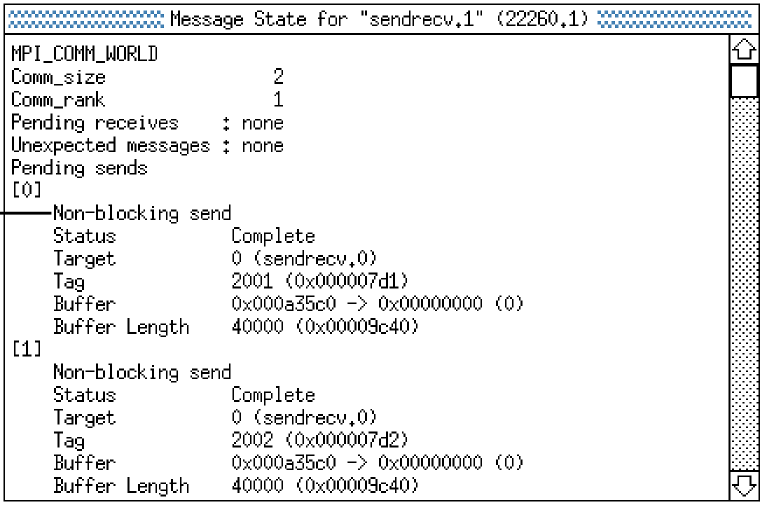


```
Message State for "sendrecv.0" (22235.1)
MPI_COMM_WORLD
Comm_size                2
Comm_rank                0
Pending receives
[0]
    Status           Pending
    Source           1 (sendrecv.1)
    Tag              2010 (0x000007da)
    User Buffer      0x000b85c0 -> 0x00000000 (0)
    Buffer Length    40000 (0x00009c40)

Unexpected messages
[0]
    Status           Complete
    Source           1 (sendrecv.1)
    Tag              2001 (0x000007d1)
    System Buffer    0x000c2208 -> 0x00000000 (0)
    Buffer Length    40000 (0x00009c40)
    Received Length  40000 (0x00009c40)
[1]
    Status           Complete
    Source           1 (sendrecv.1)
    Tag              2002 (0x000007d2)
    System Buffer    0x000cbe50 -> 0x00000000 (0)
    Buffer Length    40000 (0x00009c40)
    Received Length  40000 (0x00009c40)
```

Figure 37. Message State Unexpected Messages

Pending Send Operations

TotalView displays each pending send operations in the pending sends list. Figure 38 shows an example of MPICH pending send messages.



```
Message State for "sendrecv.1" (22260.1)
MPI_COMM_WORLD
Comm_size              2
Comm_rank              1
Pending receives     : none
Unexpected messages  : none
Pending sends
[0]
    Non-blocking send
    Status           Complete
    Target           0 (sendrecv.0)
    Tag              2001 (0x000007d1)
    Buffer           0x000a35c0 -> 0x00000000 (0)
    Buffer Length    40000 (0x00009c40)
[1]
    Non-blocking send
    Status           Complete
    Target           0 (sendrecv.0)
    Tag              2002 (0x000007d2)
    Buffer           0x000a35c0 -> 0x00000000 (0)
    Buffer Length    40000 (0x00009c40)
```

Additional information

Figure 38. Message State Pending Send Operation

The MPICH implementation does not normally maintain information about pending send operations. However at the time you configure MPICH, you can compile in additional code to maintain a list of pending send operations. These additional data structures are maintained if the program is started under control of the TotalView debugger. Otherwise they are not maintained, unless **mpirun** is passed the **–ksq** (KeepSendQueue) flag.

Depending on the device for which MPICH was configured, blocking send operations may or may not be visible. However, if they are not displayed here, you can see that these operations are taking place because the call is on the stack backtrace.

If you attach to an MPI program which is not maintaining the send queue information, the Message State display shows this message:

```
Pending sends : no information available
```

MPI Debugging Troubleshooting

If you cannot successfully start TotalView on MPI programs, check the following:

- Can you successfully start MPICH programs without TotalView? The MPICH code contains some useful scripts to help you verify that you can start remote processes on all of the machines in your machines file. (See **tstmachines** in mpich/util.)
- Does the **tvdsvr** fail to start? You must ensure that **tvdsvr** is on your PATH as it is when you log in. Remember that **rsh** is being used to start the server, and it does not pass your current environment to the process you started remotely.
- You cannot get a message queue display if you get the following warning:

  ```
  The symbols and types in the MPICH library used by
  TotalView to extract the message queues are not as
  expected in the image <<your image name>>. This is
  probably an MPICH version or configuration problem.
  ```

  You need to check the following:

  - Be sure you are using MPICH 1.1.0 or later
  - Be sure you configured it with the **–debug** flag. (To verify this, look in the **config.status** file at the root of the MPICH directory tree).
- Make sure you have the correct MPI version and you have applied the required patches. See the *TotalView Release Notes* for the most up-to-date information.
- Under some circumstances, MPICH kills TotalView with the **SIGINT** signal.You might see this behavior when you try to restart an MPICH job by using the TotalView **Delete Program** (**^Z**) command in the process window. If TotalView exits and is terminated abnormally with **Killed** message from the shell that started TotalView, try setting the TotalView **–ignore_control_c** command line option. For example:

  ```
  % setenv TOTALVIEW "totalview –ignore_control_c"

  % mpirun  –tv  /users/smith/mympichexe
  ```

Debugging PVM and DPVM Applications

You can debug applications that use the Parallel Virtual Machine (PVM) library or the Digital UNIX Parallel Virtual Machine (DPVM) library with Totalview on some platforms. TotalView supports ORNL PVM 3.3.4 or later on the Digital UNIX Alpha, Sun 4, Sun 5, RS/6000, and SGI IRIX platforms and DPVM 1.4 or later on the Digital UNIX Alpha platform.

Note: See the *TotalView Release Notes* for the most up-to-date information regarding your PVM or DPVM software.

For tips on debugging parallel applications, see “Parallel Debugging Tips,” on page 110.

Supporting Multiple Sessions

When you debug a PVM or DPVM application, TotalView becomes a PVM tasker, which establishes a debugging context for the duration of your session. You can run:

- *One* TotalView PVM or DPVM debugging session, per user, per architecture; that is, different users cannot interfere with each other on the same machine, or same machine architecture.

  One user can start TotalView to debug the same PVM or DPVM application on *different* machine architectures. However, a single user *cannot* have multiple instances of TotalView debugging the same PVM or DPVM session on a single machine architecture.

  For example, suppose you start a PVM session on a set of Sun 4 and Digital UNIX Alpha machines. In this scenario, you start two different TotalView sessions: one on a Sun 4 machine to debug the Sun 4 portion of the PVM session, and one on a Digital UNIX Alpha machine to debug the Digital UNIX Alpha portion of the PVM session. These two separate TotalView sessions (Sun 4 and Digital UNIX Alpha) do not interfere with one another.

- Similarly, in one TotalView session, one user can run either a PVM application or a DPVM application but not both.

  If you are running TotalView on a Digital Alpha, you can have two TotalView sessions, one debugging PVM and one debugging DPVM.

Setting Up ORNL PVM Debugging

To enable PVM, create a symbolic link from the PVM **bin** directory: **$HOME/pvm3/bin/$PVM_ARCH/tvdsvr** to the TotalView Debugger Server (**tvdsvr**).With this link in place, TotalView can use the **pvm_spawn()** call to spawn the debugger server tasks.

For example, if **tvdsvr** is installed in the **/opt/totalview/bin**, you can use the following command:

```
% ln –s /opt/totalview/bin/tvdsvr $HOME/pvm3/bin/$PVM_ARCH/tvdsvr
```

If the symbolic link does not exist, TotalView cannot spawn the debugger server and displays the following error:

```
Error spawning TotalView Debugger Server: No such file
```

Starting an ORNL PVM Session

Start the ORNL PVM daemon process *before* you start TotalView. See the ORNL PVM documentation for information about the PVM daemon process and console program.

1. Use the **pvm** command to start a PVM console session, which will start the PVM daemon. If PVM is not running when you start TotalView (with PVM support enabled), TotalView exits with the following message:

   `Fatal error: Error enrolling as PVM task:` *pvm error*

2. If your application uses groups, start the **pvmgs** process *before* starting TotalView. PVM groups are unrelated to TotalView process groups. For information about TotalView process groups, refer to “Examining Process Groups,” on page 129.

3. Enable PVM support in TotalView using one of the following methods:

   - With an X resource; see “totalview*pvmDebugging: {true | false},” on page 276. You need to restart TotalView after setting this new resource. For more information, refer to “X Resources,” on page 263.

   - Use command-line options to the **totalview** command:

–pvm	Enables PVM support.
–no_pvm	Disables PVM support

     The command-line options override the X resource. For more information on the **totalview** command, refer to “TotalView Command Syntax,” on page 287.

4. Set the TotalView directory search path to include the PVM directories. The list of directories must include those needed to find *both* executable and source files. The actual list of directories you need will vary, but you should always include the current directory and your home directory.

   You can set the directory search path using an X resource or the **Set Search Directory** command. Refer to "totalview*searchPath: dir1[,dir2,...]," on page 277 and "Setting Search Paths," on page 52 for more information.

   For example, to debug the PVM examples, you can specify the following list of directories in the search path:

   ```
   .
   $HOME
   $PVM_ROOT/xep
   $PVM_ROOT/xep/$PVM_ARCH
   $PVM_ROOT/src
   $PVM_ROOT/src/$PVM_ARCH
   $PVM_ROOT/bin/$PVM_ARCH
   $PVM_ROOT/examples
   $PVM_ROOT/examples/$PVM_ARCH
   $PVM_ROOT/gexamples
   $PVM_ROOT/gexamples/$PVM_ARCH
   ```

5. Verify that the default action taken by TotalView for the SIGTERM signal is appropriate. You can examine the default actions with the **Set Signal Handling Mode** command in TotalView. Refer to "Handling Signals," on page 48 for more information.

   PVM uses the SIGTERM signal to terminate processes. By default, TotalView stops a process when the process receives a SIGTERM signal, which prevents the process from being terminated. If you want the PVM process to terminate instead of stop, set the default action for the SIGTERM signal to **Resend**.

Continue with "PVM/DPVM Automatic Process Acquisition," on page 99.

Starting a DPVM Session

DPVM requires no additional user configuration. However, you must start the DPVM daemon process *before* you start TotalView. See the DPVM documentation for information about the DPVM daemon and console program.

1. Use the **dpvm** command to start a DPVM console session, which will start the DPVM daemon. If DPVM is not running when you start TotalView (with DPVM support enabled), TotalView exits with the following message:

   `Fatal error: Error enrolling as DPVM task:` *dpvm error*

2. Enable DPVM support using one of the following methods:

   - With an X resource; see“totalview*DPVMDebugging: {true | false},” on page 268. You need to restart TotalView after setting a new X resource. For more information, refer to “X Resources,” on page 263.
   - Use command-line options to the **totalview** command:

–dpvm	Enables DPVM support.
–no_dpvm	Disables DPVM support

     The command-line options override the X resource. For more information on the **totalview** command, refer to “TotalView Command Syntax,” on page 287.

3. Verify that the default action taken by TotalView for the SIGTERM signal is appropriate. You can examine the default actions with the **Set Signal Handling Mode** command in TotalView. Refer to “Handling Signals,” on page 48 for more information.

   DPVM uses the SIGTERM signal to terminate processes. By default, TotalView stops a process when the process receives a SIGTERM signal, which prevents the process from being terminated. If you want the DPVM process to terminate instead of stop, set the default action for the SIGTERM signal to **Resend**.

Note: If you enable PVM support using X resources, and you wish to use DPVM, you *must* use both **–no_pvm** and **–dpvm** command line options when you start TotalView. Similarly, if you enable DPVM support with X resources, use **–no_dpvm** and **–pvm** command line options to debug PVM. Finally, we do *not* recommend using X resources to start *both* PVM and DPVM.

PVM/DPVM Automatic Process Acquisition

This section describes how TotalView automatically acquires PVM and DPVM processes in a PVM or DPVM debugging session. Specifically TotalView uses the PVM tasker feature to intercept **pvm_spawn()** calls.

When you start TotalView as part of a PVM or DPVM debugging session, it takes the following actions:

- TotalView checks to make sure there are no other PVM or DPVM taskers running. If TotalView finds a tasker on any host that it is debugging, it exits with the message:

  ```
  Fatal error: A PVM tasker is already running on
  host 'host'
  ```

- TotalView finds all the hosts in the PVM or DPVM configuration. Using the **pvm_spawn()** call, TotalView starts a TotalView Debugger Server (**tvdsvr**) on each remote host that has the same architecture type as the host on which TotalView is running. For each debugger server that TotalView starts, it prints the following message:

  ```
  Spawning TotalView Debugger Server onto PVM host
  'host'
  ```

Note: If you add a host with a compatible machine architecture to your PVM or DPVM debugging session after you start TotalView, TotalView automatically starts a debugger server on that host.

After you start TotalView and it starts all the appropriate debugger servers, TotalView intercepts every PVM or DPVM task that is created using the **pvm_spawn()** call on the hosts that are part of the debugging session. If a PVM or DPVM task is created on a host with a different machine architecture, TotalView ignores that task.

When TotalView receives a PVM or DPVM tasker event, it takes the following actions:

1. TotalView automatically reads the symbol table of the spawned executable.
2. If a saved breakpoints file for the executable exists and you have the automatic loading of breakpoints enabled, TotalView loads the breakpoints for the process.

3. TotalView asks if you want to stop the process before it enters the **main()** routine.

   If you answer **Yes**, TotalView stops the process *before* it enters **main()** (that is before it executes any user code). This allows you to set breakpoints in the spawned process before any user code is executed. On most machine architectures, if the process is statically linked, TotalView stops it in the **start()** routine of the **crt0.o** module. If the process is dynamically linked, TotalView stops it just after it finishes running the dynamic linker. In either case, the process window displays Assembler instructions, so you need to use the **Function or File (f)** command to display the source code for the **main()** routine. For more information on this command, refer to “Finding the Source Code for Functions,” on page 116.

Attaching to PVM/DPVM Tasks

You can attach to a PVM or DPVM task, providing that the task meets the following criteria:

- The machine architecture on which the task is running is the same as the machine architecture on which TotalView is running.
- The task must be created. In the PVM tasks and configuration window, which you will learn about next, this is indicated when flag 4 is set.
- The task must not be a PVM tasker. In the PVM tasks and configuration window, this is indicated when flag 400 is clear.
- The executable name must be known. If the executable name is listed as **–**, then TotalView cannot determine the name of the executable, which can happen when a task was not created using the **pvm_spawn()** call.

To attach to a PVM or DPVM task, complete the following steps:

1. Issue the **Show All PVM Tasks (P)** command from the TotalView root window.

   The PVM tasks and configuration window is displayed, as shown in Figure 39. This window displays current information about PVM tasks and hosts, and TotalView automatically updates this information as it receives events from PVM.

Note: Since PVM does not generate all the events needed, you can use the **Update PVM Task List (u)** command to force a refresh when necessary.

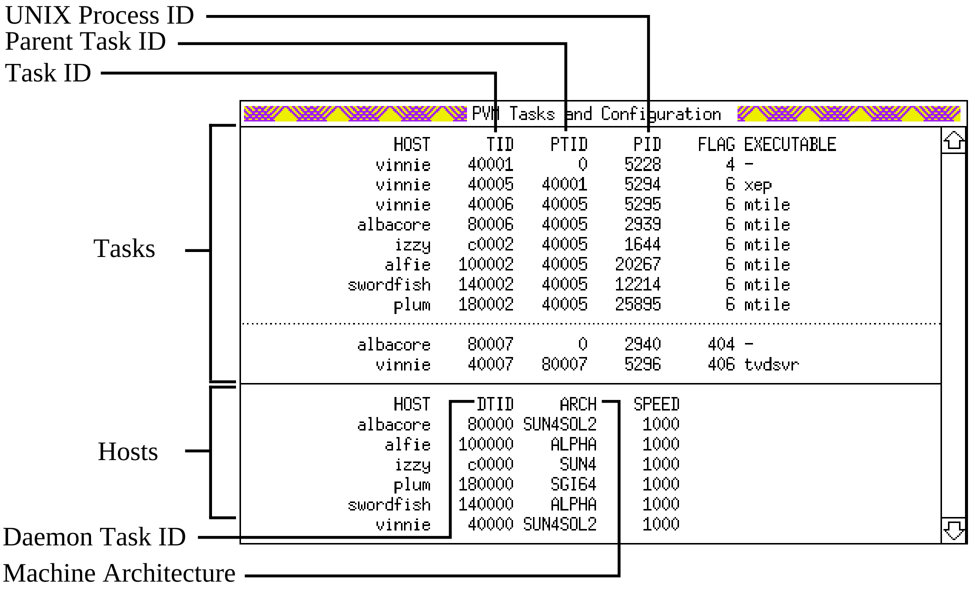
If we apply the criteria for attaching to tasks to the tasks shown in Figure 39, you can attach to the tasks named **xep** and **mtile** because they have flag 4 set, but you cannot attach to the executables named **tvdsvr** and **–** because they have flag 400 set.

2. Dive on a task entry that meets the criteria for attaching to tasks. TotalView attaches to the task.
3. If the task to which you attached has related tasks that TotalView can debug, TotalView asks if you want to attach to the relatives of the task.

   If you answer **Yes**, TotalView attaches to all the related tasks.

   If you answer **No**, TotalView attaches to only the task you dove on.

   TotalView looks for attached tasks that are related to the task to which you just attached, and if it finds any, it places them in the same program group. If TotalView is already attached to a task you dove on, TotalView simply opens and raises the process window for the task.



```
PVM Tasks and Configuration

     HOST     TID   PTID    PID   FLAG EXECUTABLE
   vinnie   40001      0   5228      4 -
   vinnie   40005  40001   5294      6 xep
   vinnie   40006  40005   5295      6 mtile
 albacore   80006  40005   2939      6 mtile
     izzy   c0002  40005   1644      6 mtile
    alfie  100002  40005  20267      6 mtile
swordfish  140002  40005  12214      6 mtile
     plum  180002  40005  25895      6 mtile
.................................................
 albacore   80007      0   2940    404 -
   vinnie   40007  80007   5296    406 tvdsvr
-------------------------------------------------
     HOST    DTID     ARCH   SPEED
 albacore   80000 SUN4SOL2    1000
    alfie  100000    ALPHA    1000
     izzy   c0000     SUN4    1000
     plum  180000    SGI64    1000
swordfish  140000    ALPHA    1000
   vinnie   40000 SUN4SOL2    1000
```

Figure 39. PVM Tasks and Configuration Window

Reserved Message Tags

TotalView uses the following PVM message tags to communicate with the PVM daemons and TotalView Debugger Server. Avoid sending messages that use these reserved tags:

0xDEB0 through 0xDEBF

Debugging Dynamic Libraries

If the set of machines in your PVM debugging session are running different versions of the same operating system, the dynamic libraries can vary from machine to machine. If this is the case, you may see strange stack backtrace results when your program is executing inside a dynamic library. To eliminate this problem, make sure all of the hosts in your PVM configuration are running the same version of the operating system and have the same dynamic libraries installed, or link your programs statically.

Cleanup of Processes

The **pvmgs** process registers its task ID in the PVM database. If the **pvmgs** process is terminated, the **pvm_joingroup()** routine hangs because PVM does not clean up the database. If this happens, you must terminate the PVM daemon and start it again.

TotalView attempts to clean up the TotalView Debugger Server daemons (**tvdsvr**), which also act as taskers, but occasionally some of these processes do not terminate. If this happens, you must manually terminate the **tvdsvr** processes.

Debugging Portland Group, Inc. (PGI) HPF Applications

TotalView allows the source level debugging of High Performance Fortran (HPF) code compiled with the Portland Group HPF (PGHPF) compiler.

Note: Debugging PGHPF programs requires a separate TotalView license key.

For tips on debugging parallel applications, see “Parallel Debugging Tips,” on page 110.

TotalView supports the following platforms:

- IBM RS/6000 and SP AIX 4.x
- SGI MIPS IRIX 6.x, for programs compiled with –64 only
- Sun Sparc SunOS 5 (Solaris 2.x)

See the *TotalView Release Notes* for supported PGHPF runtime configurations.

In addition to normal TotalView features, the TotalView PGHPF support allows:

- Source level display of HPF code
- Source level breakpoints in HPF code
- Display of distributed arrays, with optional display of the owning processor
- Visualization of distributed arrays
- Visualization of the distribution of distributed arrays
- Automatic update of all copies of replicated scalar variables

However, there are still a number of limitations:

- Display of user defined data types is not yet supported.
- **EVAL** points and expressions are executed locally and cannot reference distributed arrays (apart from the **$visualize** intrinsic, which does work).

Installing TotalView for HPF

You will need a parallel run time that TotalView understands. With TotalView 3.8 and later, and PGHPF release 2.4, TotalView can track the process start-up used by **rpm** or **smp**, the default PGHPF run time libraries. If you still want to use MPI, then you need to ensure that the MPI implementation is supported by PGHPF and TotalView. See “Debugging MPI Applications,” on page 76.

On IBM SP, or clusters of RS/6000 machines running IBM's Parallel Environment, you can use any run time library that is started using the **poe** command.

On SGI IRIX, TotalView supports 64-bit PGHPF programs only. You must compile your PGHPF program with the **–64** compiler option.

Dynamically Loaded Library

To debug PGHPF code, TotalView needs to be able to dynamically load the file **libtvhpf.so**, which is distributed as part of the PGHPF product.

TotalView searches for this file in the following order:

1. TotalView attempts to dynamically load the unadorned file name **libtvhpf.so**. This will succeed if:
    - **libtvhpf.so** is in one of the directories on your dynamic library path environment variable (LD_LIBRARY_PATH on Sun Sparc SunOS5, IBM AIX, and SGI IRIX if LD_LIBRARYN32_PATH is not set)
    - SGI IRIX only: **libtvhpf.so** is in one of the directories on your **–n32** dynamic loader path (LD_LIBRARYN32_PATH)
2. If step 1 fails, then TotalView uses the **PGI** environment variable to find the Portland Group installation tree. If the **PGI** environment variable is not set, then the default installation directory (**/usr/pgi**) is tried instead.

Depending on the target architecture, TotalView then searches the directories in the order shown in Table 8.

Table 8. PGHPF Dynamic Library Search Order

System	Search Path
IBM RS/6000 and SP AIX 4.x	$PGI/sp2/lib $PGI/rs6000/lib
Sun Sparc SunOS 5 (Solaris 2.x)	$PGI/solaris/lib

Table 8. PGHPF Dynamic Library Search Order (Continued)

System	Search Path
SGI MIPS IRIX 6.x	$PGI/sgi/lib–n32 $PGI/sgi/lib–64 $PGI/origin/lib/mips4

If all of this fails to locate a copy of **libtvhpf.so**, then, if the TotalView verbosity level is not **silent**, an error message is posted to tell you that the library could not be found, **HPF** debugging is disabled, and TotalView proceeds to debug at the intermediate Fortran level.

If you have a copy of **libtvhpf.so**, but TotalView cannot locate it using the strategy described above, then you should either move it to one of the places that will be searched by default, or add its directory to your LD_LIBRARY_PATH.

Setting Up PGHPF Compiler Defaults

Set up the HPF compiler with the correct defaults for use with MPICH, TotalView, the IBM parallel environment, and Fortran77, as in the following sections.

If you have PGHPF release 2.4, the **rc** files should already have been set up correctly, but they will use the default run time, that is, not MPI. If you want to use an MPI runtime you should consult the PGHPF manuals.

Note: With PGHPF version 2.4 and later, there is no need to use an MPICH based run time, and you can ignore this section.

Setting Up MPICH

You should follow the instructions in the PGHPF manual and MPICH manual to ensure that you can build an HPF program and run it using MPICH. One way to do this is to create your own **.pghpfrc** file and add lines similar to the following:

```
#  Set up to use my MPI with pghpf.
#  Change the path to libmpi.a as appropriate
#

INCLUDE  $DRIVER/.pghpfrc
set HPF_MPI=/where_your_mpi_lives/libmpi.a
set HPF_COMM_LIBS="-lpghpf_mpi$P  $HPF_MPI  $HPF_SOCKET"
```

Adding these lines to your **.pghpfrc** file will force **pghpf** to use the MPI communications library without requiring that you specify it on the command line at compilation time.

Setting TotalView Defaults

To debug HPF code, you will normally want to set the default behavior of breakpoints and barrier breakpoints to not stop other processes when the breakpoint is hit. For more information, refer to “Parallel Debugging Tips,” on page 110.

Other relevant HPF resources are “totalview*hpf: {true | false},” on page 271 and “totalview*hpfNode: {true | false},” on page 271.

Compiling HPF for Debugging

To compile your HPF program for use with TotalView you should use the **–g** and **–Mtotalview** flags to **pghpf** when both compiling and linking. The **–Mtv** flag is the same as the **–Mtotalview** flag.

The **–g** flag on its own produces very confusing results. You may see the HPF source code, but none of the HPF debugging features will work. If TotalView flags your HPF code in the stack backtrace as being f77, then you have probably forgotten the **–Mtv** flag when compiling.

The **–g** flag directs the PGHPF compiler to output additional information into a **.stb** file. This contains the relationship between the HPF source file and the intermediate F77 source. TotalView uses it to map HPF level entities (files, functions, variables) to the executable image.

The **–g** flag is also required on the link step, since this instructs the HPF compiler to produce a **.stx** file which indexes the external symbols in the program back to their source files. This allows TotalView to read the **.stb** files only as they are required.

If you want to debug at the level of the generated Fortran code, then you will also have to give the **–Mkeepftn** flag. Otherwise, these intermediate Fortran files are deleted by the compiler once they have been compiled.

Starting HPF Programs

The way in which an HPF parallel program is started depends on the machine on which it is running and the choice of run time library which is linked into the HPF code.

PGHPF smp and rpm libraries

To start a program linked with these libraries under TotalView control proceed as if you were using TotalView to debug the program. If you normally start the code:

```
% foo –bah –pghpf –np 6
```

you can debug it with this command:

```
% totalview foo –a –bah –pghpf –np 6
```

Starting HPF Programs with MPICH

In a workstation cluster environment using MPICH, you can debug your HPF application with TotalView by using the **–tv** flag to the **mpirun** command.

So, where you might normally run your code with the following command:

```
% mpirun –np 4 foo
```

you can invoke TotalView with the following command:

```
% mpirun –tv –np 4 foo
```

Workstation Clusters Using MPICH

Debugging workstation clusters uses the same mechanism as debugging an MPICH program, since a compiled HPF program *is* an MPICH program. For more information, refer to “Debugging MPI Applications,” on page 76.

IBM Parallel Environment

In the IBM parallel environment on an IBM SP or cluster of RS/6000 machines, parallel programs are started with the **poe** command. To debug parallel codes, you invoke TotalView on the **poe** command, for instance:

```
% totalview poe –a hpf_test –procs 6
```

For more information, refer to “Starting TotalView on a PE Job,” on page 83.

HPF TotalView Advantages

The following are the advantages of debugging HPF in TotalView:

- You can display the contents of distributed arrays by diving on the array.
- You can see the distribution of distributed arrays, for instance, onto which node a particular element of a distributed array has been mapped.
- You can update replicated scalar variables in all processes by updating the value in any process. If the values were not all the same at the start, TotalView gives you a warning, and you have to explicitly agree to the update before it will take place.
- You can export a distributed array to the TotalView visualizer the same way as any other array.

If you use the **$visualize** EVAL intrinsic, remember that EVAL code is executed by every process. Therefore, you probably want to make this an non-shared action point.

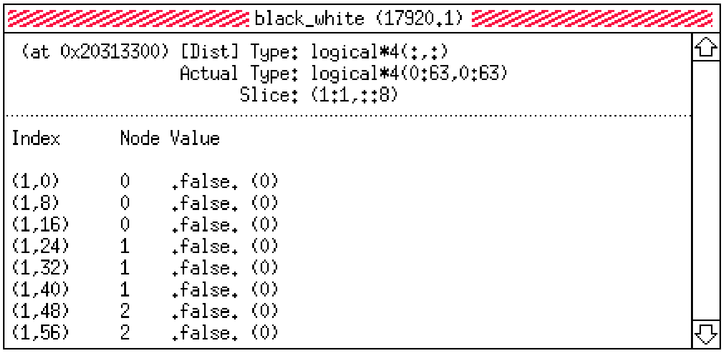
- You can export the distribution of an array to the visualizer to display it graphically.
- You see the HPF source and variables.
- You can set breakpoints in the HPF source code.

In the address display for data windows showing HPF variables, there is an additional field which tells you whether the variable is distributed [**Dist**] or replicated [**Repl**]. If you update a replicated variable, then it is updated in all the processes. A distributed variable will only be updated in its home process.

You cannot edit the address of a distributed array. If you edit the address of a replicated scalar, then it will be marked as distributed, since it no longer makes sense to update all of the processes, as you do not know what is at that address in the other processes.

When you display an HPF distributed array, TotalView can also display the logical processor on which each element resides. The display of this additional information can be changed for a single data window using the **Toggle Node Display** option in the menu of the data window. You can set the default for a whole TotalView session by using the command line options **–hpf_node** or **–no_hpf_node**, or by using the X resource “totalview*hpfNode: {true | false},” on page 271. No matter which way you set the default, you can always toggle the behavior in each window.

By default, this display is disabled. If it is enabled, then a distributed array will look like Figure 40. Otherwise, the Node column is not displayed and a distributed array display looks the same as that of a normal array.



```
black_white (17920,1)
 (at 0x20313300) [Dist] Type: logical*4(:,:)
                 Actual Type: logical*4(0:63,0:63)
                        Slice: (1:1,::8)

Index      Node Value

(1,0)      0    .false. (0)
(1,8)      0    .false. (0)
(1,16)     0    .false. (0)
(1,24)     1    .false. (0)
(1,32)     1    .false. (0)
(1,40)     1    .false. (0)
(1,48)     2    .false. (0)
(1,56)     2    .false. (0)
```

Figure 40. Block Distributed Array on Three Processes

To see the distribution of an array, or a section of an array, use the **Visualize Distribution** command from the data window menu. This command exports the HPF processor number on which each selected element of the array resides to the visualizer. This command differs from the **Visualize** command, that exports the values of the array elements, not the ownership information.

This capability is not available with the **$visualize** command, since distributions are normally static, so re-displaying them under program control does not seem to be useful.

Debugging generated FORTRAN 77

You can debug at the generated Fortran level by starting TotalView with the **–no_hpf** flag or setting the X resource **totalview*hpf** to **false**.TotalView will then ignore the **.stb** and **.stx** files and show you the generated F77. (Remember to compile with **–Mkeepftn**, or these files won't exist!). Alternatively, of course, simply removing the **.stx** file will also cause TotalView not to recognize the code as HPF.

There is no need to relink the HPF program to debug at the generated FORTRAN level.

Parallel Debugging Tips

When you are debugging your parallel programs, the following points are important to remember.

General Parallel Debugging Tips

Here are some tips that are useful for debugging most parallel programs:

- When you are debugging message-passing and other multiprocess programs, it is usually easier to understand the program's behavior if you change the default stopping action of breakpoints and barrier breakpoints. By default, when one process in a multiprocess program hits a breakpoint, TotalView will stop all the other processes. To change the default stopping action of breakpoints and barrier breakpoints:

  - Set the X resources "totalview*stopAll: {true | false}," on page 280 and/or "totalview*barrierStopAll: {true | false}," on page 266 to **false**.
  - Specify the TotalView command line options **–no_stop_all** on page 297 and/or **–no_barr_stop_all** on page 289.

  These settings cause the default breakpoint and barrier breakpoint behavior to allow other processes to continue to run when one of the processes in a group hits the breakpoint.

  These options flag only affects the default behavior. As usual, you can choose the behavior for a specific breakpoint, individually, by setting the breakpoint properties in the action points dialog box. See "Breakpoints for Multiple Processes" on page 197.

- TotalView has two features that make it easier to get all of the processes in a multiprocess program synchronized and executing the line.

  - Process barrier breakpoints and the process hold/release features work together to help you get control the execution of your processes. See "Process Barrier Breakpoints" on page 201.
  - The **Run (to selection) Group (R)** command is a special kind of single-stepping command that allows you to run a group of processes to a selected source line or instruction. See "Group-Level Single-Stepping" on page 134.

- Group commands are often more useful than process commands.
  - It is often more useful to issue the **Go Group (G)** command, from the **Go/Halt/Step/Next/Hold** submenu, to restart the whole application, rather than the **Go Process (g)** command, and to issue the **Halt Group (H)** command rather than the **Halt Process (h)** command.
  - The group-level single-stepping commands, such as **Step Group (S)** and **Next Group (N)**, allow you to single-step a group of processes in a parallel. See “Group-Level Single-Stepping” on page 134.
- If you use a process-level single-stepping command in a multiprocess program, TotalView may appear to be hung (it continuously displays the watch cursor). If you single-step a process over a statement that cannot complete without allowing another process to run, and that process is stopped, the stepping process appears to hang. In parallel programs, this happens most often if you try to single-step a process over a communication operation that cannot complete without the participation of another process. When this happens:
  - You can abort the single-step operation by pressing **Control-C** (**^C**) in any TotalView window.
  - Consider using a group-level single-step command instead.
- The TotalView root window has a feedback mechanism that helps you determine where various processes and threads are executing. When you select a line of code in the process window, the root window is updated to give you visual feedback about which processes and threads are executing that line. See “Displaying Thread and Process Locations” on page 140.
- You can view the value of a variable that is replicated across multiple processes or multiple threads in a single variable window. See “Displaying a Variable in All Processes or Threads” on page 177.
- You can restart a parallel program at any time during your debugging session. If your program runs too far, you can kill the program by displaying the **Arguments/Create/Signal** submenu in the process window and selecting the **Delete Program (^Z)** command. This command kills the master process and all the slave processes. You can then restart the master process (e.g, mpirun or poe), and all of the slave processes will be recreated. Start-up will be faster in these circumstances, because TotalView does not need to reread the symbol tables or restart its server processes as they are already running.

MPICH Specific Debugging Tips

Here are some debugging tips that apply only to MPICH:

- You can pass flags to TotalView through the MPICH **mpirun** command.

  To pass flags to TotalView when running **mpirun**, you can use the **TOTALVIEW** environment variable. For example, you can cause **mpirun** to invoke TotalView with the **–no_stop_all** flag as in the following C-shell, example:

  ```
  % setenv TOTALVIEW "totalview –no_stop_all"
  ```

- If you start remote processes with MPICH/**ch_p4**, you may need to change the way TotalView starts the servers.

  By default, TotalView uses **rsh** to start its remote server processes. This is the same behavior as **ch_p4**. If you configure MPICH/**ch_p4** to use a different start-up mechanism from another process, you will probably also need to change the way that TotalView starts the servers.

  For more information about **tvdsvr** and **rsh**, see “The Auto-Launch Feature,” on page 64. For more information about **rsh**, see “The Server Launch Command,” on page 66.

IBM PE Specific Debugging Tips

Here are some debugging tips that apply only to IBM MPI (PE):

- Avoiding unwanted time-outs

  You can cause undesired time-outs if you place breakpoints that stop other process too soon after calling **MPI_Init()** or **MPL_Init().** If you create "stop all" breakpoints, it causes the first process to get to the breakpoint to stop all the other parallel processes that have not yet arrived at the breakpoint. This may cause a timeout.

  To turn the option off, click with the right mouse button on the **stop** symbol for the breakpoint. The breakpoint dialog box will come up, in which you should deselect the box labeled “Stop All Related Processes when Breakpoint Hit.”

- Controlling the **poe** process

  The **poe** process continues under TotalView control, but normally, you should not attempt to start, stop, or otherwise interact with **poe**. The parallel tasks require that **poe** continue to run for normal functioning. For this reason, TotalView automatically continues **poe** when you continue any of the parallel tasks, if **poe** had been stopped.

- Slow processes due to node saturation

  If you try to debug a Parallel Environment for AIX program in which more than three parallel tasks are run on a single node, the parallel tasks on each such node may run noticeably slower than they would run if you weren't debugging them.

  This effect becomes more noticeable as the number of tasks increases, and, in some cases, the parallel tasks may make hardly any progress. This is because the Parallel Environment for AIX uses the SIGALRM signal to implement the communications operations, and the debugging interface in AIX requires that the debugger intercept all signals. As the number of parallel tasks on a node increases, the copy of TotalView or the TotalView Debugger Server running on that node becomes saturated, and cannot keep up with the SIGALRMs being sent, thus slowing down the tasks.

CHAPTER 6: Debugging Programs

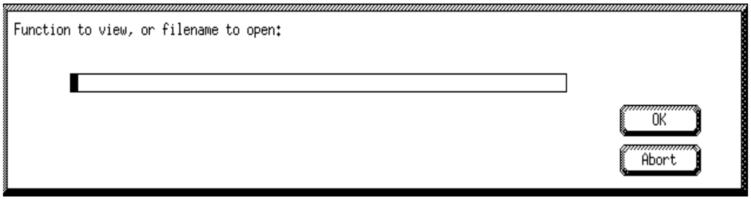
This chapter explains how to perform basic debugging tasks with TotalView. You'll learn how to:

- Find code as you debug
- Display your code in source and assembler formats
- Invoke your editor on source files you are debugging
- Return to the currently executing line in the stack frame
- Interpret status and control registers
- Use commands for controlling processes and threads
- Control process groups in multiprocess programs
- Use single-step commands
- Debug with signal handlers
- Set the program counter

Finding the Source Code for Functions

If you linked a function to your program at compile time, you can then use TotalView to search for the source code for that function. You can:

- Dive into the function name from the source code pane.
- On the **Function/File/Variable** submenu, select the **Function or File (f)** command. When prompted, type the function name in the dialog shown in Figure 41.



Function to view, or filename to open:

Figure 41. Function Name Dialog

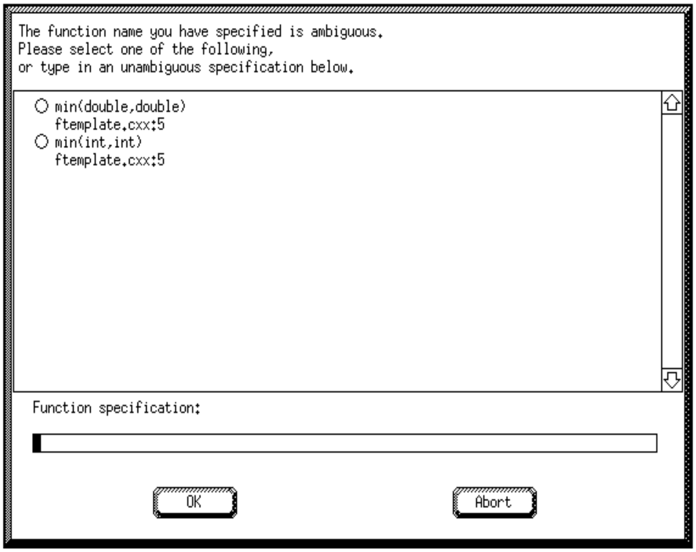
Tip: When you want to return to the original contents of the source code pane, dive into the undive icon located in the upper right corner of the source pane.

If TotalView finds the source code, it displays it in the source code pane. If the function you selected was not compiled with source line information, TotalView displays the disassembled machine code for the function instead of displaying the source code.

Tip: You can use the **Edit Source Text** command (see “Editing Source Text” on page 122 for details) or an X Window System client such as **xmore**, **vi**, or **emacs** to display these files while debugging.

Resolving Ambiguous Names

Sometimes the function name you specify is ambiguous. For example, you may have specified the name of a static function when your program contains multiple static functions by that same name. Alternatively, you may have specified the name of a member function in a C++ program where there are multiple classes with member functions of that name. Or, you may have specified the name of a template function. In all of these cases, TotalView prompts you to resolve the ambiguity. Figure 42 shows an example of the dialog that TotalView displays when it encounters an ambiguous function name.



The function name you have specified is ambiguous.
Please select one of the following,
or type in an unambiguous specification below.

○ min(double,double)
ftemplate.cxx:5
○ min(int,int)
ftemplate.cxx:5

Function specification:

Figure 42. Dialog for Resolving Ambiguous Function Names

To resolve the ambiguity, click one of the radio buttons or the text following it and then click **OK**. Alternately, you may type an unambiguous name in the Function Specification field.

Whenever you select a function name, its specification automatically appears in the **Function specification** field allowing you to create a new function specification by editing the existing one. When there are many screens of function names in the dialog, this feature lets you specify a name you want; you do not have to scroll to find a specific name.

Once the TotalView context is set to a particular instantiation of the function template, then TotalView uses that instantiation and no longer displays a dialog to disambiguate names. TotalView can prompt you to set the specific context when you:

- Specify a function name with the **Function or File (f)** command
- Dive on a name in the source pane
- Halt execution at a line in the function
- Select a function by clicking on its line in the stack trace pane
- Previously selected a line in the function and that line is still selected

Finding the Source Code for Files

You can display the source code for a given file in your program by choosing the **Function/File/Variable** submenu and selecting the **Function or File** (**f**) command. When prompted, enter the file name in the dialog box shown in Figure 41. You may enter the name of a header file if the header file contains source lines that produce executable code.

Source File Extensions

TotalView maps filename extensions to source languages as shown in Table 9.

Table 9. Source Language Mapping

File Extension	Source Language
.cxx, **.cc**, **.cpp**, **.C**, **.hxx**, **.H**	C++
.F, **.f**, **.F90**, **.f90**	FORTRAN 77 or Fortran 90
.hpf, **.HPF**	HPF
All others	C

TotalView uses one of the following methods to identify a program as FORTRAN 77 or Fortran 90:

- The compiler explicitly specifies the language in the debug information.
- The source filename has an **.f90** or **.F90** suffix; TotalView treats the program language as Fortran 90.
- The code uses Fortran 90 features such as assumed shape arrays or pointers. If TotalView determines that a file contains Fortran90 using this method, then it is possible that functions or subroutines defined earlier in the same source file will appear to be written in Fortran77. This should not be a problem, since such functions cannot be using Fortran90 features.

Examining Source and Assembler Code

In the source code pane of the process window, you can display your program in several different ways, as shown in Table 10. If you display Assembler in the source code pane, you can also display addresses in two different ways, as shown at the bottom of Table 10.

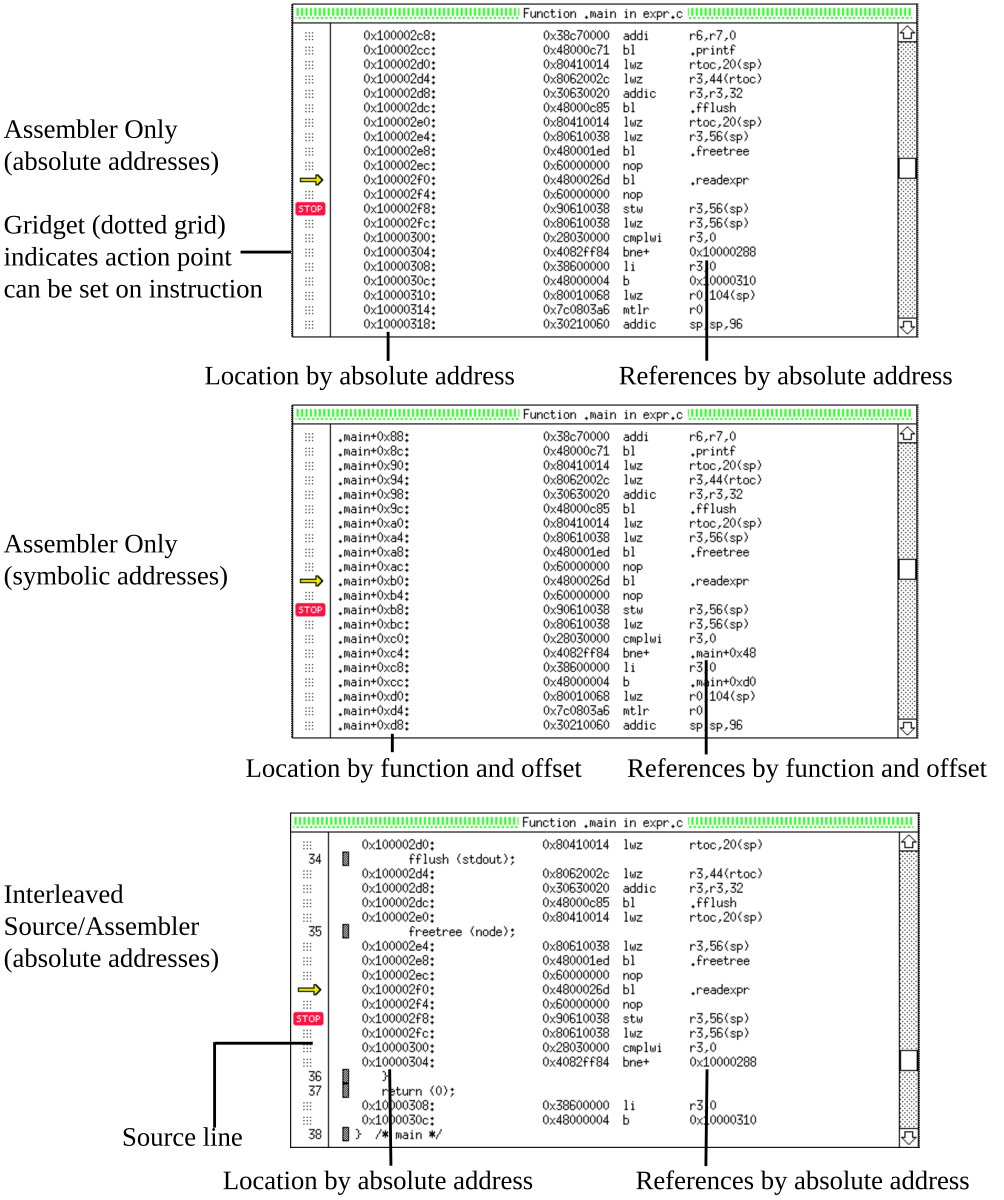
Table 10. Ways to Display Source and Assembler Code

To Display This in the Source Code Pane...	**Select This from the Display/Directory/Edit Submenu...**
Source code only (Default)	**Source Display Mode (Meta-s)**
Assembler code only	**Assembler Display Mode (Meta-a)**
Source code interleaved with Assembler code[1]	**Interleave Display Mode (Meta-i)**
Symbolic addresses (function names and offsets) for all locations and references[2]	**Display Assembler Symbolically**
Absolute addresses for all locations and references (Default)[2]	**Display Assembler by Address**

[1] Source statements are treated like comments. You can set breakpoints or evaluation points only at the machine level, not at the source level. Setting an action point at the first instruction after a source statement, however, is equivalent to setting a point at that source statement.

[2] If an address matches the address of a function, TotalView displays the function name.

Figure 43 illustrates the effect of displaying Assembler code in different ways in the source code pane. You can also display Assembler instructions in a variable window. For more information, see “Displaying Machine Instructions” on page 151.



```
Function .main in expr.c
0x100002c8:    0x38c70000  addi    r6,r7,0
0x100002cc:    0x48000c71  bl      .printf
0x100002d0:    0x80410014  lwz     rtoc,20(sp)
0x100002d4:    0x8062002c  lwz     r3,44(rtoc)
0x100002d8:    0x30630020  addic   r3,r3,32
0x100002dc:    0x48000c85  bl      .fflush
0x100002e0:    0x80410014  lwz     rtoc,20(sp)
0x100002e4:    0x80610038  lwz     r3,56(sp)
0x100002e8:    0x480001ed  bl      .freetree
0x100002ec:    0x60000000  nop
0x100002f0:    0x4800026d  bl      .readexpr
0x100002f4:    0x60000000  nop
STOP 0x100002f8:    0x90610038  stw     r3,56(sp)
0x100002fc:    0x80610038  lwz     r3,56(sp)
0x10000300:    0x28030000  cmplwi  r3,0
0x10000304:    0x4082ff84  bne+    0x10000288
0x10000308:    0x38600000  li      r3,0
0x1000030c:    0x48000004  b       0x10000310
0x10000310:    0x80010068  lwz     r0,104(sp)
0x10000314:    0x7c0803a6  mtlr    r0
0x10000318:    0x30210060  addic   sp,sp,96
```

```
Function .main in expr.c
.main+0x88:    0x38c70000  addi    r6,r7,0
.main+0x8c:    0x48000c71  bl      .printf
.main+0x90:    0x80410014  lwz     rtoc,20(sp)
.main+0x94:    0x8062002c  lwz     r3,44(rtoc)
.main+0x98:    0x30630020  addic   r3,r3,32
.main+0x9c:    0x48000c85  bl      .fflush
.main+0xa0:    0x80410014  lwz     rtoc,20(sp)
.main+0xa4:    0x80610038  lwz     r3,56(sp)
.main+0xa8:    0x480001ed  bl      .freetree
.main+0xac:    0x60000000  nop
.main+0xb0:    0x4800026d  bl      .readexpr
.main+0xb4:    0x60000000  nop
STOP .main+0xb8:    0x90610038  stw     r3,56(sp)
.main+0xbc:    0x80610038  lwz     r3,56(sp)
.main+0xc0:    0x28030000  cmplwi  r3,0
.main+0xc4:    0x4082ff84  bne+    .main+0x48
.main+0xc8:    0x38600000  li      r3,0
.main+0xcc:    0x48000004  b       .main+0xd0
.main+0xd0:    0x80010068  lwz     r0,104(sp)
.main+0xd4:    0x7c0803a6  mtlr    r0
.main+0xd8:    0x30210060  addic   sp,sp,96
```

```
Function .main in expr.c
        0x100002d0:    0x80410014  lwz     rtoc,20(sp)
34              fflush (stdout);
        0x100002d4:    0x8062002c  lwz     r3,44(rtoc)
        0x100002d8:    0x30630020  addic   r3,r3,32
        0x100002dc:    0x48000c85  bl      .fflush
        0x100002e0:    0x80410014  lwz     rtoc,20(sp)
35              freetree (node);
        0x100002e4:    0x80610038  lwz     r3,56(sp)
        0x100002e8:    0x480001ed  bl      .freetree
        0x100002ec:    0x60000000  nop
        0x100002f0:    0x4800026d  bl      .readexpr
        0x100002f4:    0x60000000  nop
STOP    0x100002f8:    0x90610038  stw     r3,56(sp)
        0x100002fc:    0x80610038  lwz     r3,56(sp)
        0x10000300:    0x28030000  cmplwi  r3,0
        0x10000304:    0x4082ff84  bne+    0x10000288
36          }
37          return (0);
        0x10000308:    0x38600000  li      r3,0
        0x1000030c:    0x48000004  b       0x10000310
38      }  /* main */
```

Assembler Only
(absolute addresses)
Gridget (dotted grid) indicates action point can be set on instruction
Location by absolute address
References by absolute address
Assembler Only
(symbolic addresses)
Location by function and offset
References by function and offset
Interleaved Source/Assembler
(absolute addresses)
Source line
Location by absolute address
References by absolute address

Figure 43. Different Ways to Display Assembler Code

Current Stack Frame

You can return to the executing line of code for the current stack frame by selecting the **Current Stackframe** (**c**) command from the **Current/Update/Relatives** submenu in the process window. This command forces the PC arrow onto the screen and discards the dive stack.

The **Current Stackframe** (**c**) command is also useful if you want to undo the effect of scrolling or finding a function or file using the **Function or File... (f)** command. For details, see “Finding the Source Code for Functions” on page 116.

If the program has not begun to run, the **Current Stackframe** (**c**) command puts you in the first executable line of code in your main program function or subroutine.

Editing Source Text

You can use the **Edit Source Text (M-e)** command on the **Display/Directory/Edit** submenu to edit source files while you are debugging. TotalView starts your editor on the source file being displayed in the source pane of the process window.

TotalView uses the editor launch string to determine how to start your editor. To change the value of the editor launch string, see “Changing the Editor Launch String” on page 122.

Changing the Editor Launch String

You can change the editor launch string to control the way the debugger starts your editor when you use the **Edit Source Text** command.

The editor launch string is processed by TotalView and expanded into a command string, that is then executed by the shell **sh**. This allows you to configure exactly how the editor is started.

TotalView recognizes certain items in the launch string, which are expanded before the debugger starts your editor. The items that are expanded are as follows:

%E Expands to the value of the EDITOR environment variable, or to **vi** if EDITOR if not set.

%N Expands to the line number in the middle of the source pane. Use this option if your editor allows you to specify an initial line number at which to position the cursor.

%S Expands to the source file name displayed in the source pane.

%F Expands to the font name with which you started TotalView.

The default editor launch string is:

```
xterm -e %E +%N %S
```

which creates an xterm window in which to run the editor. If you use an editor that creates its own X window, such as **emacs** or **xedit**, you do not need to create the xterm window, and you should change the editor launch string.

You can change the editor launch string by using one of the following methods:

- Using an X resource.

  Refer to “totalview*editorLaunchString: command_string” on page 268 for more information.

- Using the **Editor Launch String...** command on the **Display/Directory/Edit** submenu of the process window.

Interpreting Status and Control Registers

The stack frame pane in the process window lists the contents of CPU registers for the selected frame (you may need to scroll down to see them). To learn about the meaning of these registers, you need to consult the user's guide for your CPU and Appendix C, "Architectures," on page 333.

For your convenience, TotalView interprets the bit settings of certain CPU registers, such as the registers that control the rounding and exception enable modes. You can edit the values of these registers and continue execution of your program. For example, you might do this to examine the behavior of your program with a different rounding mode.

Since the registers that are interpreted vary from platform to platform, see Appendix C, "Architectures," on page 333 for information on the registers supported for your CPU. For general information on editing the value of variables (including registers), refer to "Displaying Areas of Memory" on page 150.

Starting Processes and Threads

To start a process, go to the process window and select one of the following commands from the **Go/Halt/Step/Next/Hold** submenu.

Go Process (g)
Creates and starts this process. Resumes execution if the process is not being held, already exists and is stopped or at a breakpoint. Starting a process causes all threads in the process to resume execution.

Go Group (G)
Creates and starts this process and all other processes in the multiprocess program (program group). Resumes execution and the execution of all processes in the program group if the process is not being held, already exists and is stopped or at a breakpoint.

	Note that issuing **Go Group** on a process that's already running starts the other members of the program group.
Go Thread (^G)	Starts this thread. Disabled if asynchronous thread control is not available (see "Thread-Level Control" on page 135).

For a single-process program, **Go Process** and **Go Group** are equivalent. For a single-threaded process, **Go Thread** and **Go Process** are equivalent.

Commands that contain the term Group (for example, **Go Group**) refer to all members of the program group. The term relatives generally refers to the program group as well.

Note: If a process is being held by TotalView, the above commands will not start the process or thread. See "Holding and Releasing Processes" on page 128.

Creating a Process without Starting it

The **Create Process (without starting it) (C)** command creates a process and stops it before it executes any of your program. For programs that are linked with shared libraries, TotalView allows the dynamic loader to map in shared libraries.

Creating a process without starting it is useful:

- If you want to display or change global variables after a process is created, but before it runs
- If you want to debug your C++ static constructor code

Creating a Process by Single-Stepping

The TotalView single-stepping commands allow you to create a process and run it to a certain point in your programs. The process window single-stepping commands in the **Go/Halt/Step/Next/Hold** submenu behave as follows when creating a process:

Step (source line) (s)	Creates the process and runs it to the first line of the **main()** routine.
Next (source line) (n)	Same as **Step (source line) (s)**.

Step (instruction) (i) Creates the process and instruction steps the first instruction of your program.

Next (instruction) (x) Creates the process and instruction nexts the first instruction of your program.

Run (to selection) (r) Creates the process and runs it to the line or instruction you have selected in the process window.

Stopping Processes and Threads

To stop a process or a thread, go to the process window and select one of the following commands from the **Go/Halt/Step/Next/Hold** submenu:

Halt Process (h) Stops the process.

Halt Group (H) Stops the process and all related processes.

Note that issuing **Halt Group (H)** on a process that's already stopped stops the other members of the program group.

Halt Thread (^H) Stops the thread. Disabled if asynchronous thread control is not available (see "Thread-Level Control" on page 135).

When the TotalView debugger stops a process, it updates the process window and all related windows. When you start the process again, execution continues from the point where you stopped the process.

Note: You can force the process window to update the process information using the **Update Process Info (u)** command from the **Current/Update/Relatives** submenu *without* stopping the process. TotalView will flush its internal process data cache and temporarily stop the process and reread the thread registers and memory. This allows you to quickly refresh your view of a process.

Holding and Releasing Processes

TotalView allows you to hold and release processes. When a process is held, any command that would otherwise cause the process to run, such as **Go Process (g)** or **Go Group (G)**, has no effect.

Manual hold and release are useful in a number of cases:

- If you wish to run a subset of the processes, you can manually hold all but the ones you want to run
- If a process is held at a process barrier point and you want to run it without first running all the other processes in the group to that barrier, you can release it manually and then run it

A process may also be held if it stops at a process barrier breakpoint. You can manually release a process which is being held at a process barrier breakpoint. See "Process Barrier Breakpoints" on page 201 for more information on how process barrier breakpoints interact with holding and releasing processes manually.

When a process is being held, the root window and process window display a held indicator. See Figure 80 on page 203.

To hold or release a process or group of processes:

- You can toggle the hold/release state of a process by choosing the **Hold/Release Process (w)** command from the **Go/Halt/Stop/Next/Hold** submenu in the process window.

Note: If a process is running when you issue the **Hold/Release Process (w**) command, TotalView first stops the process then holds it.

- You can hold an entire group by choosing **Hold Group** command from the **Go/Halt/Step/Next/Hold** submenu in the process window.
- You can then release the group by choosing **Release Group** command from the **Go/Halt/Step/Next/Hold** submenu in the process window.

Examining Process Groups

When you debug multiprocess programs, TotalView places processes in process groups for convenience. TotalView's process groups are not related to UNIX process groups or PVM groups in any way.

Types of Process Groups

When you start a multiprocess program, the debugger adds each process to a process group as the process starts. The debugger groups the processes depending on the type of system call (**fork()** or **execve()**) that created or changed the processes. There are two different types of process groups:

Program Group Includes the parent process and *all* related processes. A program group includes children that were forked (processes that share the same source code as the parent) and children that were forked but with a subsequent call to **execve()** (processes that do *not* share the same source code as the parent).

Share Group Includes only the related processes that share the same source code.

In general, if you are debugging a multiprocess program, the program group and share group differ only if the program has some children that are forked with a subsequent call to **execve()**.

The debugger names the processes in program groups and share groups according to the name of the source program. The parent process is named after the source program. Child processes that were forked have the same name as the parent, but with a numerical suffix (.*n*). Child processes that call **execve()** after they were forked have the parent's name, the name of the new executable (in angle brackets), and a numerical suffix.

For example, if the **generate** process forks no children, and the **filter** process forks a child process that makes a subsequent call to **execve()** to execute the **expr** program, the debugger names and groups the processes as shown in Figure 44.

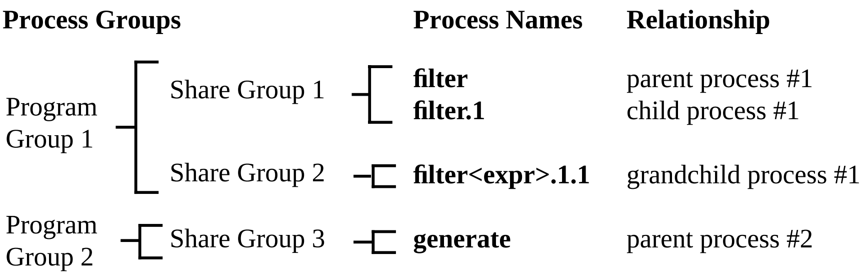
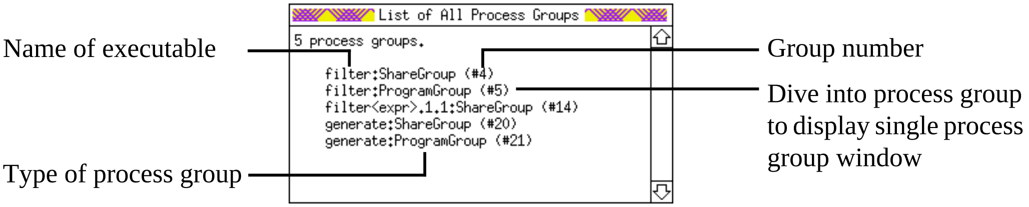


Figure 44. Example of Program Groups and Share Groups

Displaying Process Groups

The root window displays the names of individual processes in multiprocess programs, but not in the process groups. To display a list of process groups, select the **Show All Process Groups** command from the root window. The process groups window appears, as shown in Figure 45.



```
5 process groups.

    filter:ShareGroup (#4)
    filter:ProgramGroup (#5)
    filter<expr>.1.1:ShareGroup (#14)
    generate:ShareGroup (#20)
    generate:ProgramGroup (#21)
```

Name of executable

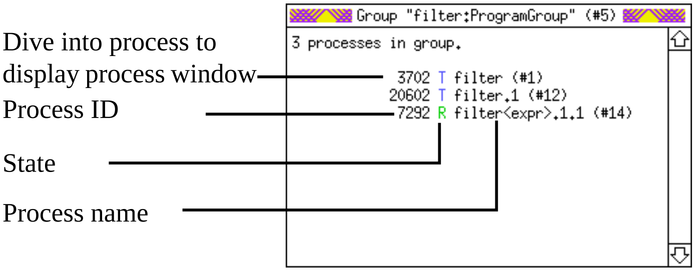
Type of process group

Group number

Dive into process group to display single process group window

Figure 45. Process Groups Window

If you dive into any process group listed in the window, a single process group window appears, as shown in Figure 46. By diving into any process listed in the window, you display the process window for the process. (You can also dive into the process listed in the root window to display its process window.)


Dive into process to display process window
Process ID
State
Process name
Group "filter:ProgramGroup" (#5)

```
3 processes in group.

     3702 T filter (#1)
    20602 T filter.1 (#12)
     7292 R filter<expr>.1.1 (#14)
```

Figure 46. Single Process Group Window

Changing Program Groups

In most situations, TotalView places a process in the correct program group, so you do not normally need to change the program group of a process.

If necessary, however, you can move processes into different program groups. When you move a process into a different program group, TotalView automatically places it in the correct share group. The advantage of moving a process into a different program group is that members of the same program group can start and stop on a breakpoint at the same time. (See “Group-Level Single-Stepping” on page 134 for details that apply to multiprocess programs.) Furthermore, members of the same share group share the same set of action points.

Note: TotalView uses the name of the executable to determine the share group to which the program belongs. TotalView does not examine the program in any way to see if it is identical to another program with the same name; TotalView assumes the programs are identical because their names are identical. Also, TotalView does not expand a program’s full pathname, so if one instance of a program is named with the full pathname (**./foo**), and another is named with the leaf name (**foo**), the programs are placed in different share groups.

To move a process into a different program group:

1. From the root window, select **Show All Process Groups**. The process groups window appears.
2. Make note of the group ID number for the *program group* into which you’re moving the process. This number is displayed in parentheses.

3. From the process window for the process to be moved, display the **Arguments/Create/Signal** submenu, and select **Set Process Program Group**. A dialog box appears, as shown in Figure 47.

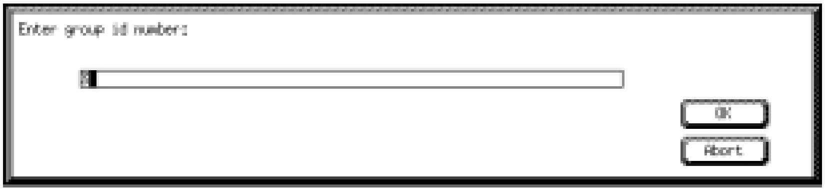


Figure 47. Dialog for Changing Process Groups

4. Enter the group ID number into the dialog box and press Return.

Finding Active Processes

Although a well-balanced multiprocess program distributes work evenly among processes, this situation does not always occur in practice. In some multiprocess programs, most of the active processes may be waiting for work. In this situation, it's tedious to look through the entire group to find the processes that are doing work. Instead, you can use the **Find Interesting Relative** command to find them quickly.

When you display the **Current/Update/Relatives** submenu and select the **Find Interesting Relative** command from the process window:

- A process group window appears, listing the processes in decreasing order of interest.
- A process window appears for the most interesting process in the group (if it does not already have a process window open).

To see additional process windows for processes in decreasing order of interest, select the **Find Interesting Relative** command again, or dive into the processes listed in the process group window.

TotalView uses the following criteria to determine the order of interest:

- Running processes are more interesting than stopped processes.
- Processes with threads at breakpoints are more interesting than those that are stopped at arbitrary locations.

- Processes with threads with deep (larger) stacks are more interesting than processes with shallow (smaller) stacks.
- Processes with threads with unusual PCs are more interesting than processes with threads with identical PCs. (The debugger examines all the threads and produces a histogram of their PCs to determine this.)

Setting a Breakpoint

You can set breakpoints in your program by selecting the boxed line numbers in the source code pane of a process window. A boxed line number indicates that the line generates executable code. A **STOP** icon masking a line number indicates that there is a breakpoint set on the line. Selecting the **STOP** icon clears the breakpoint.

When a program reaches a breakpoint it stops. You can let the program resume execution in any of the following ways:

- Use single-step commands described in “Single-Stepping” on page 133.
- Use a signal handler if your program contains one to continue with a specific signal. See “Continuing with a Specific Signal” on page 142.
- Use the set program counter command to resume execution at a specific source line, machine instruction, or absolute hexadecimal value. See “Setting the Program Counter” on page 143.
- Set breakpoints at lines you choose and allow your program to execute to that breakpoint. See “Setting Breakpoints” on page 190.
- Set conditional breakpoints that cause a program to stop after it evaluates a condition that you define, for example “stop when a value is less than 8.”See “Defining Evaluation Points” on page 205.

TotalView provides additional features for working with breakpoints, process barrier breakpoints, and evaluation points. For more information, refer to Chapter 8, “Setting Action Points,” on page 187.

Single-Stepping

TotalView supports single stepping commands that allow you to do any of the following:

- Execute one source line or machine instruction at a time
- Step over or into function calls
- Run to a selected line, which acts like a temporary breakpoint
- Run until a function call returns

Single-step commands are on the **Go/Halt/Step/Next/Hold** submenu of the process window, and operate at one of three levels: process-level, group-level or thread-level. The various levels affect which threads within a process and processes within a group are allowed to run while the single-stepping command is executing.

In all cases, the single-step commands operate on the *primary thread*, which is the thread that is selected in the current process window.

Process-Level Single-Stepping

The process-level single-step commands step the primary thread and allow other threads in the process to run. Threads that reach the stopping point in advance of the primary thread resume execution. The primary thread must reach the stopping point before execution stops.

Some operating systems only implement a synchronous run model; when one thread in the process runs for any reason, all threads must run. To step a thread on these systems, you must use the full-process, single step commands. These platforms include: LynxOS, IRIX, and SunOS operating systems.

Group-Level Single-Stepping

The group-level single-step commands operate on a TotalView process group (the program and share groups described on page 129). When you issue the command, TotalView identifies the processes and threads that are *similar* to the primary process and thread. These processes form a *step* group; TotalView steps this group and stops only when *all* its members come to the command stopping point. Similar processes are in the same share group (they execute the same code) and have at least one thread with a PC that matches the PC of the primary thread. When several threads in a process are similar to the primary thread, TotalView arbitrarily assigns one thread to the step group.

Membership in the step group can change while a group single-step command executes. A thread can leave the step group if its PC diverges from that of the primary thread, for example if it executes a conditional branch that moves away from the primary thread. A process and thread that are not included in the step group at command onset, can synchronize execution with the primary process. TotalView then includes these cases in the step group.

The **Run (to selection) Group (R)** command does not work like the other group single-step commands. It stops when the primary thread and at least one thread from each process in the share group reach the command stopping point. This allows you to use the command to synchronize a group of processes and bring them to one location.

Thread-Level Single-Stepping

The thread-level single-step commands step the primary thread to the command stopping point, while holding other user threads in the process stopped.

Note: When it can identify manager threads, TotalView runs them as it steps the single thread. Otherwise, TotalView runs the primary thread by itself.

Beware that the thread-level single step operations can fail to complete if the primary thread depends on the input or output of a thread that is not running. For example, if the primary thread requires a lock that another thread holds, and step over a call that tries to acquire the lock, then the primary thread cannot continue successfully. The other thread has to be allowed to run in order to release the lock.

Thread-Level Control

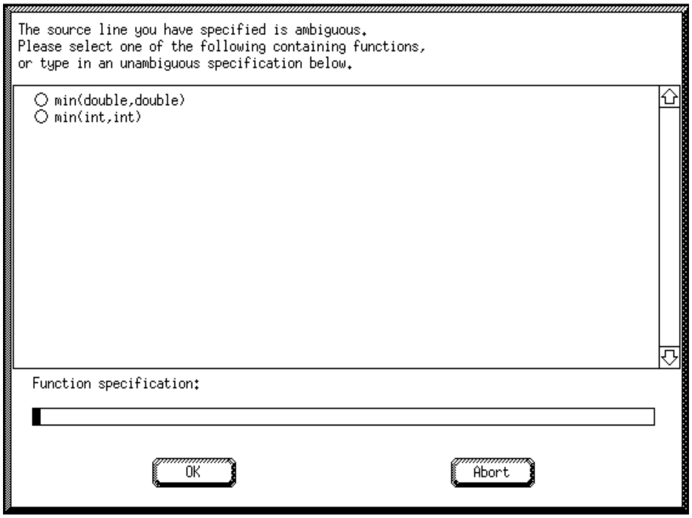
Only some operating systems allow a single thread to start and stop independently of others in the same process (this is known as asynchronous thread control). TotalView single-thread commands are operable only on the Sun4 OS, Sun5 OS, Alpha Digital UNIX, and IBM AIX operating systems.

Selecting Source Lines

Several of the single-stepping commands require you to select a line or machine instruction in the source pane of the process window. To select a source line, simply position the cursor over the desired line and select it. To deselect a source line, select it again.

Note: See "Displaying Thread and Process Locations" on page 140 for a description of the side effect selecting a line or machine instruction has on the root window display.

If you select a source line that has more than one instantiation (for example, in a C++ function template or code in a header file), TotalView prompts you to select a specific instantiation as shown in Figure 48.



The source line you have specified is ambiguous.
Please select one of the following containing functions,
or type in an unambiguous specification below.

○ min(double,double)
○ min(int,int)

Function specification:

Figure 48. Dialog for Resolving Ambiguous Source Lines

To use this dialog box:

1. Select the function instantiation you want, or type in the function specification.
2. Select the OK button.
3. Use the Abort button to abort setting the source line selection.

Single-Step Commands

To execute a single-step command first select a thread, and then select a single-step command from the **Go/Halt/Step/Next/Hold** submenu in the process window.

The following applies to all single step command:

- To cancel any single-step command in progress, position the mouse pointer in the process window and press CTRL-C.
- If your program reaches a breakpoint while stepping over a function, TotalView cancels the operation and your program stops at the breakpoint.
- If you issue a source line step command and the primary thread is executing in a function that has no source line information, TotalView performs the corresponding instruction step instead.

Stepping Into Functions Calls

To execute a single source line or instruction, and possibly step into a function call, select one of the following commands:

Step (source line) (s)	Executes a single source line at the process-level, stepping into functions, if any.
Step (source line) Group (S)	Executes a single source line at the group-level, stepping into functions, if any.
Step (source line) Thread (M-^s)	Executes a single source line at the thread-level, stepping into functions, if any.
Step (instruction) (i)	Executes a single machine instruction at the process-level, stepping into functions, if any.
Step (instruction) Group (I)	Executes a single machine instruction at the group-level, stepping into functions, if any.
Step (instruction) Thread (M-^i)	Executes a single machine instruction at the thread-level, stepping into functions, if any.

Using these commands, if you single-step a source line that contains a function call, you automatically step into the function, if there is source line information available for it. If desired, you can single-step over a function call as described in the next section.

Stepping Over Function Calls

When you step over a function, TotalView stops execution when the primary thread returns from the function and reaches the source line or instruction after the function call. To step over a function call select one of the following commands:

Next (source line) (n) Executes a single source line at the process-level, stepping over functions, if any.

Next (source line) Group (N) Executes a single source line at the group-level, stepping over functions, if any.

Next (source line) Thread (M-^n) Executes a single source line at the thread-level, stepping over functions, if any.

Next (instruction) (x) Executes a single machine instruction at the process-level, stepping over functions, if any.

Next (instruction) Group (X) Executes a single machine instruction at the group-level, stepping over functions, if any.

Next (instruction) Thread (M-^x)

Executes a single machine instruction at the thread-level, stepping over functions, if any.

Executing to a Selected Line

You don't have to set a breakpoint to stop execution on a specific line. TotalView provides a convenient way for you to run your program to a selected line or machine instruction. To do so, complete these steps from the process window:

1. In the source code pane, select the source line or instruction on which you want the program to stop execution.
2. Select one of the following commands:

Run (to selection) (r) Runs the process until the primary thread reaches the selected line.

Run (to selection) Group (R) Runs the primary thread and all the processes in the share group until it and at least one thread from each process in the share group reach the selected line. Allows you to synchronize a group of processes and bring them to one location.

Run (to selection) Thread (M-^r) Runs the primary thread until it reaches the selected line.

You can also run to a selected line in a nested stack frame. To do so:

1. Select a nested frame in the stack trace pane.
2. Select a source line or instruction within the function.
3. Issue a **Run (to selection)** command.

TotalView executes the primary thread until it reaches the selected line in the selected stack frame.

If your program calls recursive functions, you can select a nested stack frame in the stack trace pane to tailor execution even more. In this situation, TotalView uses the frame pointer (FP) of the selected stack frame and the selected source line or instruction to determine when to stop execution. When your program reaches the selected line during execution, TotalView compares the value of the selected FP to the value of the current FP in the following way:

- If the value of the current FP is deeper (more deeply nested) than the value of the selected FP, TotalView automatically continues your program.
- If the value of the current FP is equal or shallower (less deeply nested) than the value of the selected FP, TotalView stops your program.

Executing to the Completion of a Function

You can single-step your program out of a function call. To finish executing the current function in a thread, select one of the following commands:

Return (out of function) (o)
Runs the process until the primary thread returns from the current function.

Return (out of function) Group (O)
Runs the primary thread and all the processes in the share group until the primary thread returns from the current function.

Return (out of function) Thread (M-^r)
Runs the primary thread until it returns from the current function.

When the command completes, the primary thread is left stopped at the instruction after the one that called the function.

You can also return out of several functions at once. To do so:

1. Select a nested stack frame in the stack trace pane.
2. Issue a **Return (out of function)** command.

TotalView executes the primary thread until it returns *to* the function in the selected frame.

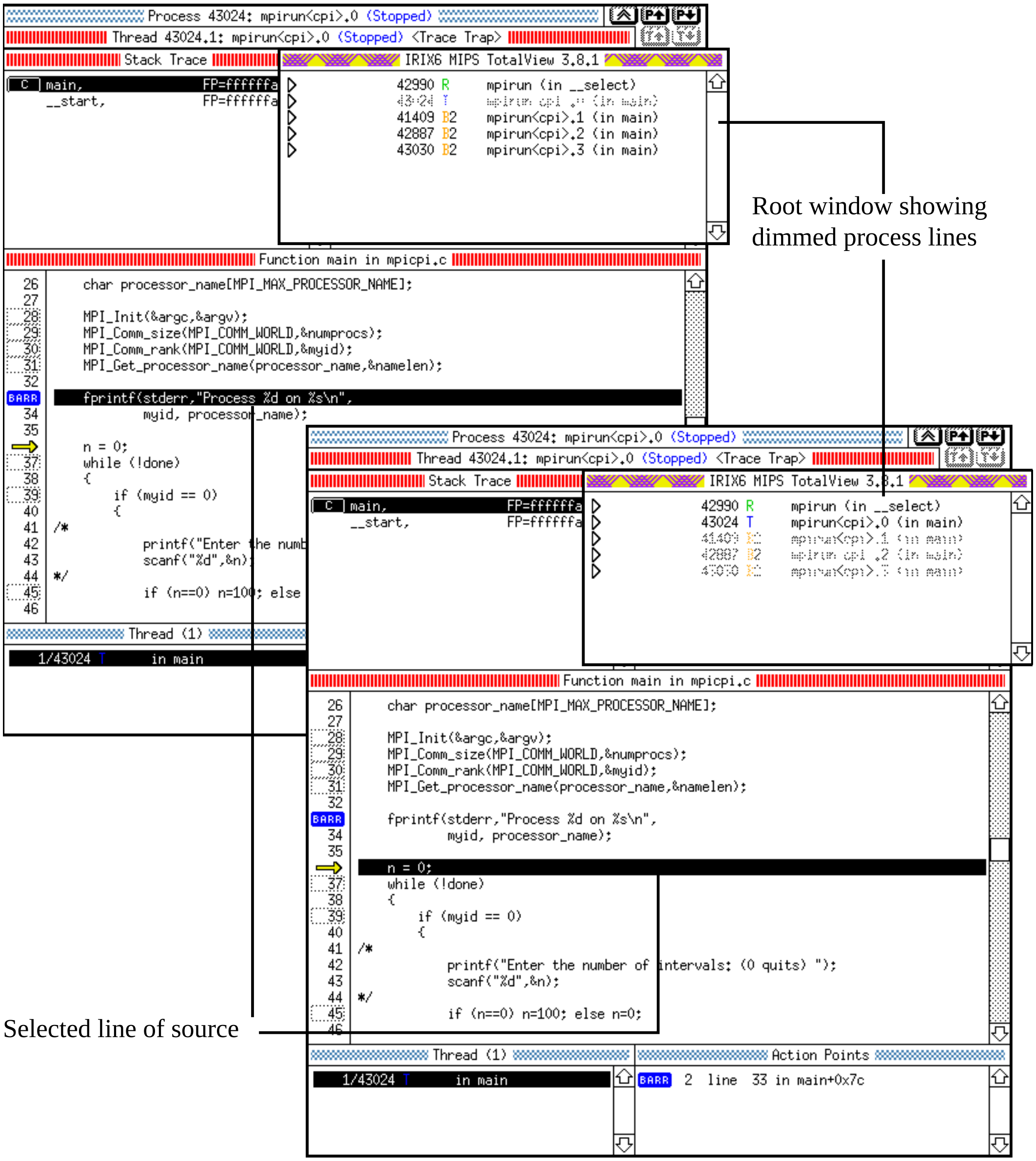
If your program calls recursive functions or mutually recursive functions, you can select a nested stack frame in the stack trace pane to tailor completion of the function even more. In this situation, TotalView uses the frame pointer (FP) of the selected stack frame and the selected source line or instruction to determine when to stop execution. When your program reaches the selected line, TotalView compares the value of the selected FP with the value of the current FP in the following way:

- If the value of the current FP is deeper (more deeply nested) than the value of the selected FP, TotalView automatically continues your program.
- If the value of the current FP is equal or shallower (less deeply nested) than the value of the selected FP, TotalView stops your program. If your program reaches a breakpoint while executing to a selected line, TotalView cancels the operation and your program stops at the breakpoint.

Displaying Thread and Process Locations

You can see which processes and threads in the share group are at a particular location by selecting a source line or machine instruction in the source pane of the process window. TotalView dims thread and process information in the root window if the thread or process is not at the selected line. A process is considered at the selected line if any of the threads in the process are at that line. Selecting a line in the process window that is already selected, will remove the dimming in the root window.

The root window reflects the line that you selected *most recently*. If you have several process windows open, the display in the root window will change depending on the line you selected last in a process window.The display can also change after an operation that changes the process state, or when you issue an **Update Process Info (u)** command.



Process 43024: mpirun<cpi>.0 (Stopped)

Thread 43024.1: mpirun<cpi>.0 (Stopped) <Trace Trap>

Stack Trace

```
C  main,       FP=ffffffa
   __start,    FP=ffffffa
```

IRIX6 MIPS TotalView 3.8.1

```
42990 R    mpirun (in __select)
43024 T    mpirun<cpi>.0 (in main)
41409 B2   mpirun<cpi>.1 (in main)
42887 B2   mpirun<cpi>.2 (in main)
43030 B2   mpirun<cpi>.3 (in main)
```

Function main in mpicpi.c

```
    char processor_name[MPI_MAX_PROCESSOR_NAME];

    MPI_Init(&argc,&argv);
    MPI_Comm_size(MPI_COMM_WORLD,&numprocs);
    MPI_Comm_rank(MPI_COMM_WORLD,&myid);
    MPI_Get_processor_name(processor_name,&namelen);

    fprintf(stderr,"Process %d on %s\n",
            myid, processor_name);

    n = 0;
    while (!done)
    {
        if (myid == 0)
        {
/*
            printf("Enter the numb
            scanf("%d",&n);
*/
            if (n==0) n=100; else
```

Thread (1)

```
1/43024 T    in main
```

Process 43024: mpirun<cpi>.0 (Stopped)

Thread 43024.1: mpirun<cpi>.0 (Stopped) <Trace Trap>

Stack Trace

```
C  main,       FP=ffffffa
   __start,    FP=ffffffa
```

IRIX6 MIPS TotalView 3.8.1

```
42990 R    mpirun (in __select)
43024 T    mpirun<cpi>.0 (in main)
41409 B2   mpirun<cpi>.1 (in main)
42887 B2   mpirun<cpi>.2 (in main)
43030 B2   mpirun<cpi>.3 (in main)
```

Function main in mpicpi.c

```
    char processor_name[MPI_MAX_PROCESSOR_NAME];

    MPI_Init(&argc,&argv);
    MPI_Comm_size(MPI_COMM_WORLD,&numprocs);
    MPI_Comm_rank(MPI_COMM_WORLD,&myid);
    MPI_Get_processor_name(processor_name,&namelen);

    fprintf(stderr,"Process %d on %s\n",
            myid, processor_name);

    n = 0;
    while (!done)
    {
        if (myid == 0)
        {
/*
            printf("Enter the number of intervals: (0 quits) ");
            scanf("%d",&n);
*/
            if (n==0) n=100; else n=0;
```

Thread (1)

```
1/43024 T    in main
```

Action Points

```
BARR  2  line  33 in main+0x7c
```

Root window showing dimmed process lines

Selected line of source

Figure 49. Dimmed Process Information in the Root Window

Figure 49 shows root windows with dimmed process information and the corresponding process windows that create this output. In this example, the parallel program was run to a barrier breakpoint, and one process (**mpirun<cpi>.0**) was single-stepped to the next source line. In the top, half of the figure, the line of source at the barrier breakpoint in the process window was selected. The root window shows the processes at that line not dimmed, and one process not at that line dimmed. In the bottom half of the figure, the line at which the one process is stopped was selected. The one process (**mpirun<cpi>.0**) was not dimmed, but the others were dimmed. Finally, since the MPI starter process (**mpirun**) is not in the same share group as the processes running the **cpi** program, the process information is subject to dimming.

Continuing with a Specific Signal

Continuing execution of your program with a specific signal can be useful if your program contains a signal handler. To do so, complete these steps from the process window:

1. Display the **Go/Halt/Step/Next/Hold** submenu and select the **Set Continuation Signal** command.
2. In the dialog box, enter the name (such as SIGINT) or number (such as 2) of the signal to be sent to the thread.
3. Select OK.

Note: The continuation signal is set for the thread you are focused on in the process window. If the target operating system supports the multithreaded signal delivery capability, you may set a separate continuation signal for each thread. If this capability is not supported, then this command will clear any continuation signal you specified for other threads in the process.

4. Continue execution of your program with the **Go**, **Step**, **Next,** or **Detach from Process** command.

   TotalView continues the thread(s) with the specified signal(s).

Setting the Program Counter

You might find it useful to resume the execution of a thread at some statement other than the one where it stopped. To do this, you reset the value of the program counter (PC). For example, you might want to skip over some code, execute some code again after changing certain variables, or restart a thread that is in an error state.

Setting the program counter can be crucial when you want to restart a thread that is in an error state. Although the PC icon in the tag field points to the source statement that caused the error, the PC actually points to the *failed machine instruction* within the source statement. You need to explicitly reset the PC to the beginning of the source statement. (You can verify the actual location of the PC before and after resetting it by displaying it in the stack frame pane or displaying interleaved source and Assembler code in the source code pane.)

In TotalView, you can set the PC of a stopped thread to a selected source line, a selected instruction, or an absolute value (in hexadecimal). When you set the PC to a selected line, the PC points to the memory location where the statement *begins*. For most situations, setting the PC to a selected line of source code is sufficient.

To set the PC to a selected line:

1. If you need to set the PC to a location somewhere *within* a line of source code, display the Assembler code. To do so, display the **Display/Directory/Edit** submenu and select the **Interleave Display Mode (M-i)** command.
2. Select the source line or instruction in the source code pane. TotalView highlights the line in reverse video.

   If you select a line in a C++ function template that has more than one instantiation, you will be prompted to select the instantiation that you want. See the section “Executing to a Selected Line” on page 138 for a description of how this works.

3. Display the **Go/Halt/Step/Next/Hold** submenu and select the **Set PC to Selection... (p)** command. TotalView asks for confirmation, resets the PC, and moves the PC icon to the selected line.

When you select a line and ask the debugger to set the PC to that line, TotalView attempts to force the thread to continue execution at that statement *in the currently selected stack frame*. If the currently selected stack frame is not the top stack frame, the debugger asks your permission to unwind the stack:

```
This frame is buried. Should we attempt to unwind
the stack?
```

If you select **Yes**, the debugger *discards* all deeper stack frames (that is, all stack frames that are more deeply nested than the selected stack frame) and resets the machine registers to the proper value for the selected frame. If you select **No**, the debugger sets the PC to the selected line, but it leaves the stack and registers in their current state. Since you cannot assume that the stack and registers have correct values, selecting **No** can cause problems. We recommend that you select **Yes**.

In general, we only recommend setting the PC to an absolute address for very advanced users. If you need to do this, make sure you have the correct address; *no* verification is done.

To set the PC to an absolute address:

1. Display the **Go/Halt/Step/Next/Hold** submenu and select the **Set PC to Absolute Value...** command. A dialog box prompts you for a hexadecimal address.
2. Enter the hexadecimal address into the dialog box.
3. Select **OK**. The debugger resets the PC and moves the PC arrow to the line containing the absolute address.

Deleting Processes

To delete a process or group of processes, display the **Arguments/Create/Signal** submenu and select the **Delete Program (^Z))** command. If the process is part of a multiprocess program, the debugger deletes all related processes as well. The next time you start the process, for example, by using the **Go Process (g)** command, the debugger creates and starts a fresh process.

Restarting Programs

You can use the **Restart Program** command to restart a program that is running or one that is stopped but has not exited. To restart a program, choose **Restart Program** from the **Arguments/Create/Signal** submenu in the process window.

If the process is part of a multi-process program, TotalView deletes all related processes, restarts the master process, and runs the newly created program.

Note: The Restart Program command is equivalent to the **Delete Program (^Z)** command followed by the **Go Process (g)** command.

CHAPTER 7: Examining and Changing Data

This chapter explains how to examine and change data as you debug your program. You'll learn how to:

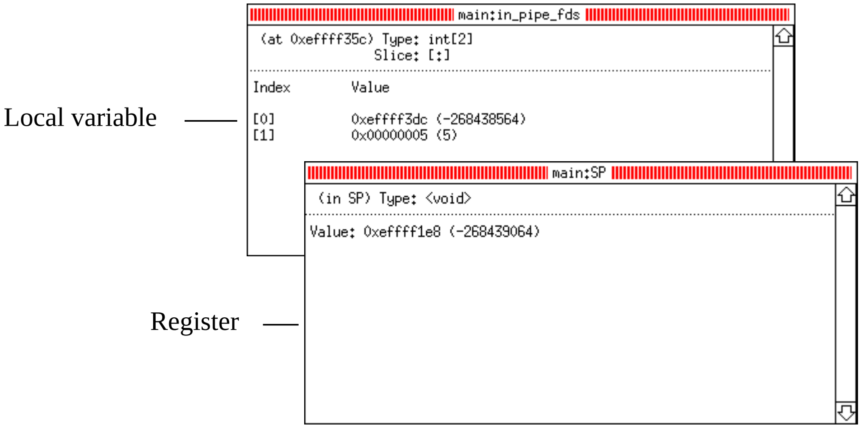
- Display variable windows
- Dive into variables
- Change the values of variables
- Change the data types of variables
- Display machine instructions
- Change the addresses of variables
- Display C++ and Fortran types
- Display array slices
- Display the value of a variable in all processes or threads
- Visualize array data
- Display mutexes
- Display conditional variables

Displaying Variable Windows

You can display variable windows for local variables, registers, global variables, areas of memory, and machine instructions.

Displaying Local Variables and Registers

In the stack frame pane of the process window, you can dive into any formal parameter, local variable, or register to display a variable window. You can also dive into formal parameters and local variables in the source code pane. The variable window lists the name, address, data type, and value for the object, as shown in Figure 50.



```
main:in_pipe_fds
 (at 0xeffff35c) Type: int[2]
             Slice: [:]
Index       Value

[0]         0xeffff3dc (-268438564)
[1]         0x00000005 (5)
```

```
main:SP
 (in SP) Type: <void>
Value: 0xeffff1e8 (-268439064)
```

Figure 50. Diving into Local Variables and Registers

You can also display a local variable using the **Variable... (v)** command of the **Function/File/Variable** submenu in the process window. When prompted, enter the name of the variable in the dialog box.

If you keep the variable windows open while you continue to run the process or thread, the debugger updates the information in the windows when the process or thread stops for any reason. When TotalView is unable to find a stack frame for a local variable that is currently displayed, **Stale** appears in the pane header to warn you that you cannot trust the data, since no such variable exists.

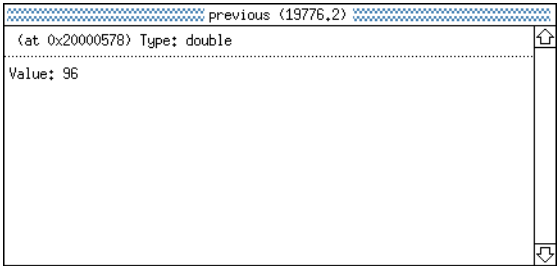
Note: When you debug recursive code, TotalView does not automatically refocus a data pane onto the leaf invocation of a recursive function. If you have a breakpoint in a recursive function, you might need to explicitly open a new data pane to see the value of a local variable for that stack frame. This is so, even though there is a window that shows the same variable in the same function for a higher invocation.

Displaying a Global Variable

You can display a global variable in two different ways:

- Diving into the variable in the source code pane.
- Displaying the **Function/File/Variable** submenu and selecting the **Variable... (v)** command. When prompted, enter the name of the variable in the dialog box.

A variable window appears for the global variable, as shown in Figure 51.

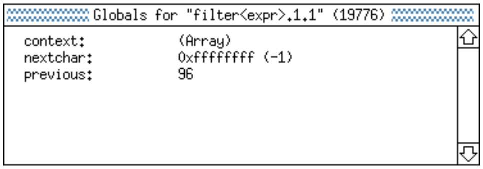


```
(at 0x20000578) Type: double
Value: 96
```

Figure 51. Variable Window for a Global Variable

Displaying All Global Variables

For convenience, you can display *all* global variables used by the current process. To do so, display the **Function/File/Variable** submenu and select the **Global Variables Window (V)** command. A global variables window appears listing the name and value of every global variable used by the process, as shown in Figure 52.



```
Globals for "filter<expr>.1.1" (19776)
context:          (Array)
nextchar:         0xffffffff (-1)
previous:         96
```

Figure 52. Global Variables Window

If desired, you can display a variable window for any global variable listed in the global variables window. To do so, either:

- Dive into the variable in the global variables window.
- Select the **Variable... (v)** command from the global variables window, and enter the name of the variable in the dialog box.

Displaying Areas of Memory

You can display areas of memory in hexadecimal and decimal. To do so, display the **Function/File/Variable** submenu and select the **Variable... (v)** command. When prompted, enter one of the following in the dialog box:

- A hexadecimal address

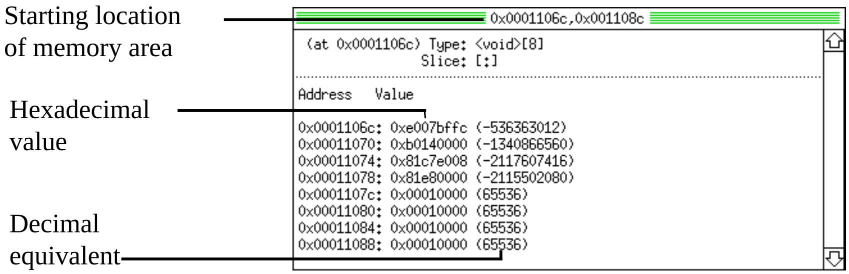
  When you enter a single address, the debugger displays the word of data stored at that address.

- A range of hexadecimal addresses

  When you enter a range of addresses, the debugger displays the data (in word increments) between the first and last address. To enter a range of addresses, enter the first address, a comma (**,**), and the last address.

Note: All hexadecimal addresses must have the “0x” prefix.

The variable window for an area of memory, shown in Figure 53, displays the address and contents of each word increment.



Starting location of memory area

Hexadecimal value

Decimal equivalent

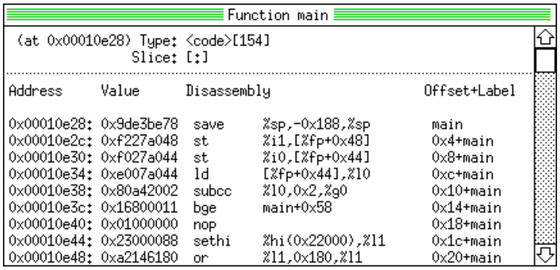
```
0x0001106c,0x001108c
 (at 0x0001106c) Type: <void>[8]
                Slice: [:]

Address   Value

0x0001106c: 0xe007bffc (-536363012)
0x00011070: 0xb0140000 (-1340866560)
0x00011074: 0x81c7e008 (-2117607416)
0x00011078: 0x81e80000 (-2115502080)
0x0001107c: 0x00010000 (65536)
0x00011080: 0x00010000 (65536)
0x00011084: 0x00010000 (65536)
0x00011088: 0x00010000 (65536)
```

Figure 53. Variable Window for Area of Memory

Displaying Machine Instructions

You can display the machine instructions for entire routines in the following ways:

- Dive into the address of an Assembler instruction in the source code pane (such as **main+0x10** or **0x60**). A variable window displays the instructions for the entire function and highlights the instruction that you dived into.
- Dive into the PC in the stack frame pane. A variable window lists the instructions for the entire function containing the PC, and highlights the instruction to which the PC points, as shown in Figure 54.



```
Function main
 (at 0x00010e28) Type: <code>[154]
                Slice: [:]

Address     Value      Disassembly                    Offset+Label

0x00010e28: 0x9de3be78  save     %sp,-0x188,%sp        main
0x00010e2c: 0xf227a048  st       %i1,[%fp+0x48]        0x4+main
0x00010e30: 0xf027a044  st       %i0,[%fp+0x44]        0x8+main
0x00010e34: 0xe007a044  ld       [%fp+0x44],%l0        0xc+main
0x00010e38: 0x80a42002  subcc    %l0,0x2,%g0           0x10+main
0x00010e3c: 0x16800011  bge      main+0x58             0x14+main
0x00010e40: 0x01000000  nop                            0x18+main
0x00010e44: 0x23000088  sethi    %hi(0x22000),%l1      0x1c+main
0x00010e48: 0xa2146180  or       %l1,0x180,%l1         0x20+main
```

Figure 54. Variable Window with Machine Instructions

- Cast a variable to type **<code>**, as described in “Changing Type Strings to Display Machine Instructions” on page 163

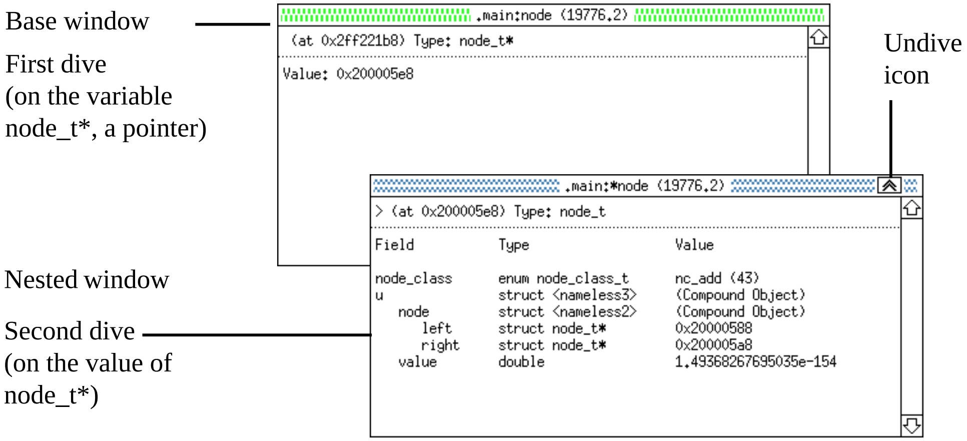
Closing Variable Windows

When you are finished analyzing the information in a variable window, you can issue the **Close Window (q)** command (to close the window) or the **Close All Similar Windows (Q)** command (to close *all* variable windows).

Diving in Variable Windows

If the variable you display in a variable window is a pointer, structure, or array, you can dive into the contents listed in the variable window. This additional dive is called a *nested dive*. When you perform a nested dive, the variable window replaces the original information with information about the current variable. With nested dives, the original variable window is known as the *base window*.

Figure 55 shows the results of diving into a variable in the stack frame pane of **main()** in the process window. In this example, we dove into a variable named **node** with a type of **node_t***, which is a pointer. The first variable window (base window) in the figure displays the value of **node**.



```
 (at 0x2ff221b8) Type: node_t*

Value: 0x200005e8
```

```
> (at 0x200005e8) Type: node_t

Field          Type                 Value

node_class     enum node_class_t    nc_add (43)
u              struct <nameless3>   (Compound Object)
   node        struct <nameless2>   (Compound Object)
      left     struct node_t*       0x20000588
      right    struct node_t*       0x200005a8
   value       double               1.49368267695035e-154
```

Figure 55. Nested Dives

Then, we dove on the value shown in the base window, and a nested dive window replaced it. The nested dive window is shown at the bottom of the figure; it shows the structure referenced by the **node_t*** pointer.

Also, notice that the number of right angle brackets (>) in the upper left hand corner indicates the number of nested dives that were performed in the window. TotalView maintains each dive as part of a dive stack.

You can manipulate variable windows and nested dive windows in the following ways:

- To “undive” from a nested dive, you click the Dive mouse button on the undive icon, and the previous contents of the variable window appears.
- If you have performed several nested dives and want to create a new base window, select the **New Base Window** command from the variable window.
- If you dive into a variable that already has a variable window open, the variable window pops to the surface. If you want a duplicate variable window open, hold down the Shift key when you dive on the variable.
- If you select the **Duplicate Window** command from the variable window, a new variable window appears that is a duplicate of the current variable window except that it has an empty dive stack.

Changing the Values of Variables

You can change the value of any variable or the contents of any memory location by completing these steps in the variable window:

1. Select the value and use the field editor to change the value as desired.

   You can type an expression as the value, including logical operators, if desired. For example, you can enter 1024*1024.

2. Press Return to confirm your changes.

You can also edit the value of variables directly from the stack frame pane by selecting them.

Note: You cannot change the value of bit fields directly, however, you can use the expression window to assign a value to a bit field. See “Evaluating Expressions” on page 215.

Changing the Data Type of Variables

The data type that you declared for the variable determines its *format* and *size* (amount of memory) in the variable window. For example, in C, if you declare an **int** variable, the debugger displays the variable as an integer.

You can change the way data is displayed in the variable window by editing the data type. This is known as type casting. TotalView assigns type strings to all data types, and in most cases, they are identical to their programming language counterparts.

- When displaying a C variable, TotalView type strings are identical to C type representations, except for pointers to arrays. By default, TotalView uses a simpler syntax for pointers to arrays.
- When displaying a Fortran variable, TotalView type strings are identical to Fortran type representations for most data types, including INTEGER, REAL, DOUBLE PRECISION, COMPLEX, LOGICAL, and CHARACTER.

To change a type string in a variable window:

> Using the field editor, edit the type string in the type field for the window. If the window contains a structure with a list of fields, you can edit the type strings of the fields listed in the window.

Note: When you edit a type string, the TotalView debugger changes how it displays the variable in the current variable window, but other windows listing the variable remain the same.

How TotalView Displays C Data Types

TotalView's syntax is identical to C cast syntax for all data types except pointers to arrays. Thus, you use C cast syntax for **int**, **unsigned**, **short**, **float**, **double**, **union**, and all named **struct** types.

You read TotalView type strings from right to left. For example, **<string>*[20]*** is a pointer to an array of 20 pointers to **<string>**.

Table 11 shows some common type strings.

Table 11. Common Type Strings

Type String	Meaning
int	Integer
int*	Pointer to integer
int[10]	Array of 10 integers
<string>	Null-terminated character string
<string>**	Pointer to a pointer to a null-terminated character string
<string>*[20]*	Pointer to an array of 20 pointers to null-terminated strings

The following sections comment on some of the more complex type strings.

If You Prefer C Cast Syntax

If desired, you can always enter C cast syntax *verbatim* in the type field for any type, and the debugger will understand it. In addition, the debugger can display C cast syntax permanently if you set an X Window Resource. See "totalview*cTypeStrings: {true | false}" on page 267 for further information.

Pointers to Arrays

Suppose you declared a variable **vbl** as a pointer to an array of 23 pointers to an array of 12 objects of type **mytype_t**. To *declare* the variable in your C program, you use the syntax:

```
mytype_t (*(*vbl)[23]) [12];
```

To *cast* **vbl** to the same type in your C program:

```
(mytype_t (*(*)[23])[12])vbl
```

TotalView's type string syntax for **vbl** would be:

```
mytype_t[12]*[23]*
```

Arrays

Array type names can include a lower and upper bound separated by a colon.

By default, the lower bound for a C or C++ array is 0, and the lower bound for a Fortran array is 1. In the following example, an array of integers is declared in C and then in Fortran:

```
int a[10]
```

```
integer a(10)
```

In the C example, the elements of the array range from **a[0]** to **a[9]**, while in the Fortran example, the elements of the array range from **a(1)** to **a(10)**.

When the lower bound for an array dimension is the default for the language, TotalView displays only the extent (that is, the number of elements) of the dimension. Consider the following array declaration in Fortran:

```
integer a(1:7,1:8)
```

Since both dimensions of the array use the default lower bound for Fortran (1), TotalView displays the data type of the array using only the extent of each dimension, as follows:

```
integer(7,8)
```

In the case where an array declaration does not use the default lower bound, TotalView displays both the lower bound and upper bound for each dimension of the array. For example, in Fortran, an array of integers with the first dimension ranging from –1 to 5 and the second dimension ranging from 2 to 10 is declared as follows:

```
integer a(-1:5,2:10)
```

TotalView displays the following data type for this Fortran array:

```
integer(-1:5,2:10)
```

When you edit a dimension of an array in TotalView, you can enter just the extent (if using the default lower bound) or both the lower and upper bounds separated by a colon.

If desired, you can display a subsection of an array. Refer to “Displaying Array Slices” on page 172 for further information.

Typedefs

The debugger recognizes the names defined with **typedef**, but displays the *definition* of such a type (that is, the base data type), rather than the name. For example, if you declared the following:

```
typedef double *dptr_t;
dptr_t p_vbl;
```

The debugger displays the type string for **p_vbl** as **double***, not as **dptr_t**.

Structures

For structures, the debugger treats the string **struct** as a keyword. You can type **struct** in as part of the type string, but it is optional. If you have a structure and another data type with the same name, you must include **struct** with the name of the structure so the debugger can distinguish between the two data types.

If you name a structure using **typedef**, the debugger uses the **typedef** name as the type string. Otherwise, the debugger uses the structure tag for the **struct**.

For example, consider the structure definition:

```
typedef struct mystruc_struct {
   int field_1;
   int field_2;
} mystruc_type;
```

The debugger displays **mystruc_type** as the type string for **struct mystruc_struct**.

The debugger does not understand actual structure definitions in the type string. For example, the debugger does not understand the type string **struct {int a; int b;}**.

Unions

The debugger displays a union as it does a structure. Even though the fields of a union are overlaid in storage, the debugger displays them on separate lines in the variable window.

Note: When the TotalView debugger displays some complex arrays and structures, it displays the (Compound Object) or (Array) type strings in the variable window. Editing the (Compound Object) or (Array) type strings might yield undesirable results. We do not recommend editing these type strings.

Built-In Type Strings

TotalView provides a number of predefined types. These types are enclosed in angle brackets to avoid conflict with types already defined in the language. You can use these built-in types anywhere a user-defined type can be used, such as in an expression. These types are also useful when debugging executables with no debugging symbol table information. Table 12 lists the built-in types.

Table 12. Built-In Type Strings

Type String	Language	Size	Meaning
<string>	C	**char**	Array of characters
<void>	C	**long**	Area of memory
<code>	C	parcel[1]	Machine instructions
<address>	C	**void***	Void pointer (address)
<char>	C	**char**	Character
<short>	C	**short**	Short integer
<int>	C	**int**	Integer
<long>	C	**long**	Long integer
<long long>	C	**long long**	Long long integer
<float>	C	**float**	Single-precision floating-point number
<double>	C	**double**	Double-precision floating-point number
<extended>	C	**long double**	Extended-precision floating-point number[2]
<character>	Fortran	**character**	Character
<integer>	Fortran	**integer**	Integer
<integer*1>	Fortran	**integer*1**	One-byte integer
<integer*2>	Fortran	**integer*2**	Two-byte integer

Table 12. Built-In Type Strings (Continued)

Type String	**Language**	**Size**	**Meaning**
<integer*4>	Fortran	**integer*4**	Four-byte integer
<integer*8>	Fortran	**integer*8**	Eight-byte integer
<logical>	Fortran	**logical**	Logical
<logical*1>	Fortran	**logical*1**	One-byte logical
<logical*2>	Fortran	**logical*2**	Two-byte logical
<logical*4>	Fortran	**logical*4**	Four-byte logical
<logical*8>	Fortran	**logical*8**	Eight-byte logical
<real>	Fortran	**real**	Single-precision floating-point number
<real*4>	Fortran	**real*4**	Four-byte floating-point number
<real*8>	Fortran	**real*8**	Eight-byte floating-point number
<real*16>	Fortran	**real*16**	Sixteen-byte floating-point number
<double precision>	Fortran	**double precision**	Double-precision floating-point number
<complex>	Fortran	**complex**	Single-precision floating-point complex number[3]
<complex*8>	Fortran	**complex*8**	real*4-precision floating-point complex number[4]
<complex*16>	Fortran	**complex*16**	real*8-precision floating-point complex number[5]

[1] A parcel is defined to be the number of bytes required to hold the shortest instruction for the target architecture.

[2] Extended-precision numbers must to be supported by target architecture.

[3] **complex** types contain a Real_Part and an Imaginary_Part, which are both of type **real**.

[4] **complex*8** types contain a Real_Part and an Imaginary_Part, which are both of type **real*4**.

[5] **complex*16** types contain a Real_Part and an Imaginary_Part, which are both of type **real*8**.

The following sections give more detail about several of the built-in types.

Character arrays (<string> data type)

If you declare a character array as **char vbl[*n*]**, the debugger automatically changes the type to **<string>[*n*]**, a null-terminated, quoted string with a maximum length of *n*. Thus, by default, the array is displayed as a quoted string of *n* characters, terminated by a null character. Similarly, the debugger changes **char*** declarations to **<string>*** (a pointer to a null-terminated string).

Since many character arrays in C are indeed strings, the debugger's **<string>** type string can be very convenient. If, however, you intended the **char** data type to be a pointer to a single character or an *array* of characters, you can edit the **<string>** back to a **char** (or **char[*n*]**) to display the variable as you declared it.

Areas of memory (<void> data type)

The debugger uses the **<void>** type string for data of an unknown type, such as the data contained in registers or in an arbitrary block of memory. The **<void>** type string is similar to the **int** in the C language.

If you dive into registers or display an area of memory, the debugger lists the contents as a **<void>** data type. Further, if you display an array of **<void>** variables, the index for each object in the array is the *address*, not an integer. This address can be useful when you display large areas of memory.

If desired, you can change a **<void>** type string to any other legal type. Likewise, you can change any legal type into a **<void>** to see the variable in hexadecimal.

Instructions (<code> data type)

The debugger uses the **<code>** data type to display the contents of a location as machine instructions. Thus, to look at disassembled code that is stored at any location, dive on the location and change the type string to **<code>**. To specify a block of locations, use **<code>[*n*]**, where *n* is the number of locations to be displayed.

Type Casting Examples

This section contains some common type casting examples.

Example: Displaying the argv Array

Typically, you declare **argv**, the second argument passed to your **main()** routine, as either a **char **argv** or **char *argv[ ]**. Since these declarations are equivalent (a pointer to one or more pointers to characters), the debugger converts both to the type **<string>**** (a pointer to one or more pointers to null-terminated strings).

Suppose **argv** points to an array of *20* pointers to character strings. To edit the type string (**<string>****) so that the debugger displays the array of 20 pointers:

1. Select the type string for **argv**.
2. Edit the type string using the field editor commands. Change it to:

   <string>*[20]*

3. To display the array, dive into the value field for **argv**.

Example: Displaying Declared Arrays

You can display declared arrays in the same way you display local and global variables. In the stack frame or source code pane, dive into the declared array. A variable window displays the elements of the array.

Example: Displaying Allocated Arrays

C code uses pointers for dynamically allocated arrays. For example, consider the following:

```
      int *p = malloc(sizeof(int) * 20);
```

In this example, TotalView doesn’t know that **p** actually points to an array of integers. To display the array:

1. Dive on the variable of type **int***.
2. Change its type to **int[20]***.
3. Dive on the value of the pointer to display the array of 20 integers.

Opaque Type Definitions

An opaque type is a data type that is not fully specified. For example the following C declaration defines **p** with a type of pointer to opaque **struct foo**:

```
struct foo;
struct foo *p;
```

When TotalView encounters type information that indicates a type is opaque, it enters the type into the type table with **<opaque>** appended to the type name. With the previous example, TotalView enters the following type name in the type table:

```
struct foo <opaque>
```

If the type is opaque and another module defines the type fully, then you can delete **<opaque>** from the data type to have TotalView find the real definition for the type.

On the platforms where TotalView uses lazy reading of the symbol table, you must force TotalView to read the symbols from the module containing the full type definition of the opaque type. Use the Function or File command to force TotalView to read the symbols, as described in "Finding the Source Code for Functions" on page 116.

Changing the Address of Variables

You can edit the *address* of a variable in a variable window. When you edit the address, the variable window shows the contents of the new location.

You can also enter an address expression, such as **0x10b8–0x80**.

Changing Type Strings to Display Machine Instructions

You can display machine instructions in any variable window. To do so:

1. Select the type string at the top of the variable window.
2. Change the type string to be an array of **<code>** data types, where the number of elements, *n*, indicates the number of instructions to be displayed:

   <code>[*n*]

   The debugger displays the contents of the current variable, register, or area of memory, as machine-level instructions.

The variable window (shown in Figure 54 on page 151) lists the following information about each machine instruction:

Address The machine address of the instruction.

Value The hexadecimal value stored in the location.

Disassembly The instruction and operands stored in the location.

Offset+Label The symbolic address of the location as a hexadecimal offset from a routine name.

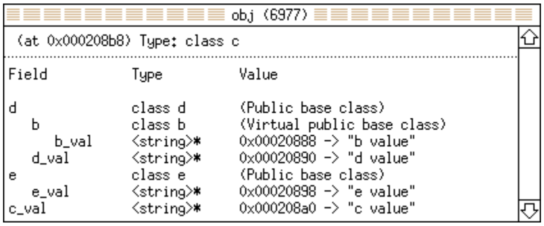
You can also edit the value listed in the **value** field for each machine instruction.

Displaying C++ Types

Classes

TotalView displays C++ classes and accepts the string **class** as a keyword. When you debug C++, TotalView also accepts the unadorned name of a class, struct, union, or enum in the type field. TotalView displays nested classes showing the derivation by indentation. For example, Figure 56 shows how TotalView displays a **class c**, defined as follows:

```
class b {
  char * b_val;
 public:
  b() {b_val = “b value“;} };

class d : virtual public b {
  char * d_val;
public:
  d() {d_val = “d value“;} };

class e {
  char * e_val;
 public:
  e() {e_val = “e value“;} };

class c : public d, public e {
  char * c_val;
 public:
  c() {c_val = “c value“;} };
```



```
 (at 0x000208b8) Type: class c

Field          Type          Value

d              class d       (Public base class)
   b           class b       (Virtual public base class)
      b_val    <string>*     0x00020888 -> "b value"
   d_val       <string>*     0x00020890 -> "d value"
e              class e       (Public base class)
   e_val       <string>*     0x00020898 -> "e value"
c_val          <string>*     0x000208a0 -> "c value"
```

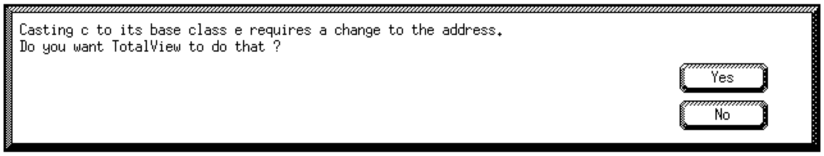
Figure 56. Displaying Nested C++ Classes

Note: Some C++ compilers do not output accessibility information. In these cases, the information is omitted from the display.

Changing Class Types in C++

Based on the C++ derivation hierarchy for a class, TotalView tries to display the correct data when you change the type of a data pane to move up or down the derivation hierarchy.

If a change in the data type also requires a change in the address of the data that is currently displayed, TotalView queries you about changing the address. For example, if you edit the type field in the **class c** shown in Figure 56 to **class e**, TotalView queries as shown in Figure 57:

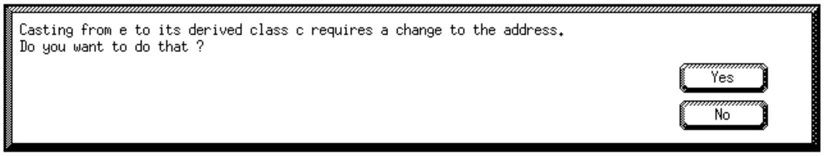


Casting c to its base class e requires a change to the address.
Do you want TotalView to do that ?

Figure 57. C++ Type Cast to Base Class Dialog Box

If you answer yes, TotalView changes the data and address to ensure that it displays the correct base class member. If you answer no, then TotalView displays the area of store as though it is an instance of the type you cast to, but the address is unchanged.

Similarly, if you change a data type in the data pane in order to cast a base class to a derived class, and that change requires a change to the address, TotalView asks you to confirm the operation. For example, Figure 58 show the dialog posted if we cast the from **class e** to **class c**:



Casting from e to its derived class c requires a change to the address.
Do you want to do that ?

Figure 58. C++ Type Cast to Derived Class Dialog Box

Displaying Fortran Types

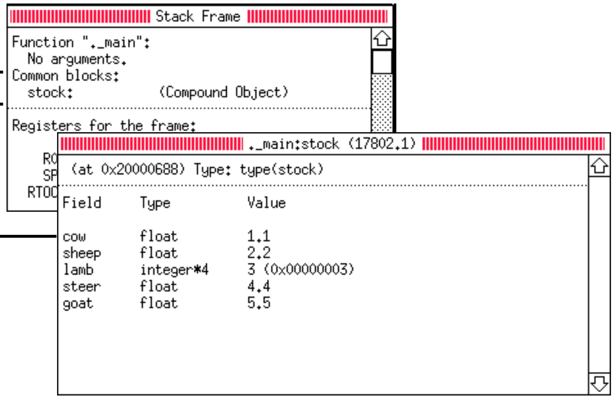
TotalView allows you to display FORTRAN 77 and Fortran 90 data types.

Displaying Fortran Common Blocks

TotalView handles Fortran common blocks in a manner consistent with the semantics of Fortran. The names of common block members have function scope, not global scope.

For each common block that is defined within the scope of a subroutine or function, TotalView creates an entry in that function’s common block list. The stack frame pane in the process window displays the name of each common block for a function.

TotalView creates a user defined data type for the common block. in which each of the common block members are fields in the type. If you dive on a common block name in the stack frame pane, TotalView displays the entire common block in a variable window, as shown in Figure 59.



Common block list in stack frame pane

Dive on common block to see elements

```
Stack Frame
Function "._main":
  No arguments.
Common blocks:
  stock:           (Compound Object)

Registers for the frame:

    RO
    SP
  RTOC
```

```
._main:stock (17802,1)
 (at 0x20000688) Type: type(stock)

Field      Type         Value

cow        float        1.1
sheep      float        2.2
lamb       integer*4    3 (0x00000003)
steer      float        4.4
goat       float        5.5
```

Figure 59. Diving into Common Block List in Stack Frame Pane

If you dive on a common block member name, TotalView searches all the common blocks for a matching member name and displays the member in a variable window.

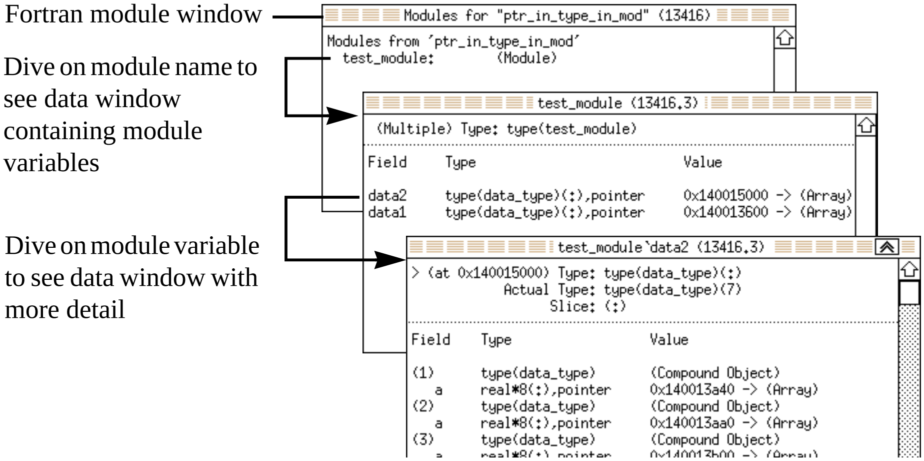
Normally, TotalView displays the initial address for a common block in the data pane. When the common block is a composite object with separate addresses for each component, TotalView displays the **Multiple** tag to indicate that there is no single address that can be given for the value of the address of the whole object.

Displaying Fortran Module Data

TotalView tries to locate all of the data associated with a given Fortran module and provide a single display that contains all of it. For functions and subroutines defined in the module, TotalView adds the full module data definition to the list of modules displayed in the stack frame pane.

For functions that *use* a module it is often not possible to determine from the debug information either that a module has been used, or what the true names of the variables in the module were. In this case (depending on what debug information is available), module variables either appear as local variables of the subroutine, or a module appears on the list of modules in the stack frame pane that contains (with appropriate renaming) only the variables used by the subroutine.

Alternatively, you can view a list of all of the modules of which TotalView is aware by using the **Fortran Modules Window** (**M**) command from the F**unction/File/Variable** submenu. This window behaves like the **Global Variables** window, so you can dive through an entry to display the actual module data. See Figure 60.


Modules for "ptr_in_type_in_mod" (13416)
test_module (13416.3)
test_module`data2 (13416.3)

Fortran module window

Dive on module name to see data window containing module variables

Dive on module variable to see data window with more detail

```
Modules from 'ptr_in_type_in_mod'
  test_module:        (Module)
```

```
 (Multiple) Type: type(test_module)

Field    Type                           Value

data2    type(data_type)(:),pointer     0x140015000 -> (Array)
data1    type(data_type)(:),pointer     0x140013600 -> (Array)
```

```
> (at 0x140015000) Type: type(data_type)(:)
          Actual Type: type(data_type)(7)
                Slice: (:)

Field    Type                 Value

(1)      type(data_type)      (Compound Object)
   a     real*8(:),pointer    0x140013a40 -> (Array)
(2)      type(data_type)      (Compound Object)
   a     real*8(:),pointer    0x140013aa0 -> (Array)
(3)      type(data_type)      (Compound Object)
   a     real*8(:),pointer    0x140013b00 -> (Array)
```

Figure 60. Fortran Modules Window

Note: SUNPro compiler users: it is not possible to find out which modules exist in a program without reading all of the debug information for the program. To display module data, you must ensure that the debug information for a file that contains the module definition or a module function has been read.

Debugging Fortran 90 Modules

In Fortran 90 or 95, you can place functions, subroutines, and variables inside modules. These modules can then be USEd by other compilation units to include the definitions from the module.

When modules are USEd, the names in the module become available in the using compilation unit, unless they have been excluded by USE, ONLY, or renamed. This means that from the Fortran source code you do not need to explicitly name a module function or variable.

However when debugging in TotalView, you may want to view the source for a *specific* function that exists in a module, and whose name is also used as a function in other modules. Similarly, when looking at a stack backtrace, it is important to know which of the functions has actually been called. To make this clear, TotalView uses the syntax:

modulename`functionname

when it displays a function from a module. You can use this syntax explicitly in the “**Function or File** (**f**)” command in the **Function/File/Variable** menu.

Fortran 90 also introduced the idea of a contained function that is only visible in the scope of its parent and siblings. Once again, there can be many contained functions in a program, all using the same name. TotalView uses a similar syntax to disambiguate these functions. If the compiler mangled the function name for the nested function, TotalView displays it with this syntax:

parentfunction()`containedfunction

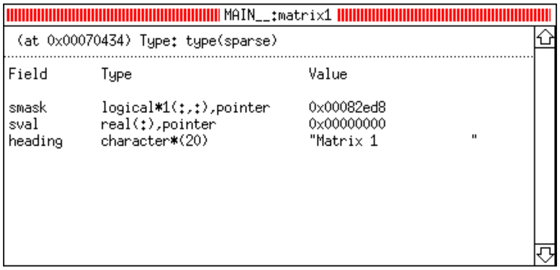
If you give an ambiguous name for a function, then TotalView displays dialog showing all of the possible matching functions. See “Finding the Source Code for Functions” on page 116 for more information.

Within contained functions, all of the variables in the parent function are visible and accessible via a static chain. If the compiler has passed on information about the static chain, TotalView can access these variables in the same way as the compiled code does, and they will be visible in data panes, and from evaluation points or expressions. If the compiler does not report about the static chain, then TotalView can find these up-level variables and display then in data panes, but you will not be able to use them in evaluation points or expressions.

F90 User Defined Type

A Fortran 90 user defined type is similar to a C structure. TotalView displays a user defined type as **type(***name***)**, which is the same syntax used in Fortran 90 to create a user defined type. For example, a variable of **type(***sparse***)**, declared as in the following code fragment, appears in Figure 61.

```
type sparse
   logical*1, pointer :: smask (:,:)
   real, pointer      :: sval (:)
   character (20)     :: heading
end type sparse
```



```
MAIN__:matrix1
 (at 0x00070434) Type: type(sparse)

Field       Type                        Value

smask       logical*1(:,:),pointer      0x00082ed8
sval        real(:),pointer             0x00000000
heading     character*(20)              "Matrix 1           "
```

Figure 61. Fortran 90 User Defined Type

F90 Deferred Shape Array Type

Fortran90 allows you to define deferred shape arrays and pointers. The actual bounds of the array are not determined until the array is allocated, the pointer is assigned to, or, in the case of an assumed shape argument to a subroutine, the subroutine is called. The type of deferred shape arrays is displayed by TotalView as *type***(:)**, in the same way that such an array would be declared in Fortran.

When TotalView displays the data for a deferred shape arrays, it displays both the type used in the definition of the variable, and the actual type that this instance of the variable has. The actual type is not editable, since you can achieve the same effect by editing the type.

The type of a deferred shape rank 2 array of REAL data with runtime lower bounds of -1 and 2, and upper bounds of 5 and 10 is shown in the following example:

```
       Type: real(:,:)
Actual Type: real(-1:5,2:10)
      Slice: (:,:)
```

F90 Pointer Type

A Fortran 90 pointer type allows you to point to scalar or array types. TotalView displays pointer types as *type***,pointer**, which is the syntax used in Fortran 90 to create a pointer variable.

For example, a **pointer** to a rank 1 deferred shape array of **real** data will be displayed in the variable window as:

```
Type: real(:),pointer
```

To view the data itself, you must dive on the value.

Note: With the IBM xlf compiler, TotalView cannot determine the rank of the array from the debug information. In this case, the type of a pointer to an array will appear as "**type(...),pointer**". The actual rank will be filled in correctly when you dive through the pointer to look at the data.

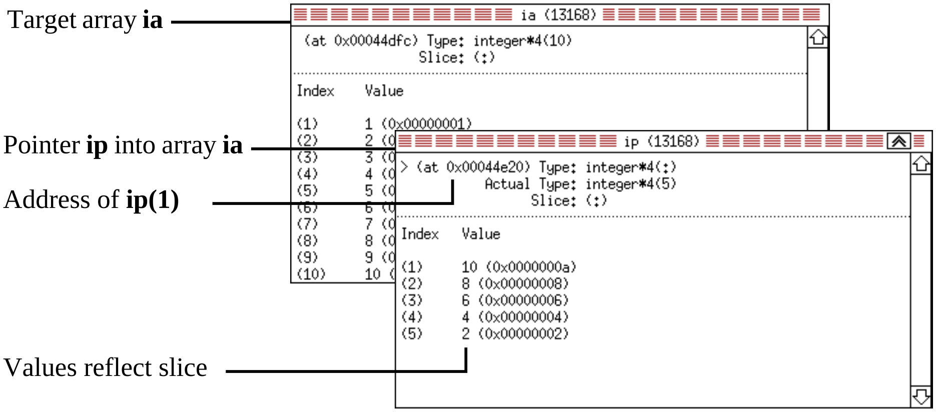
The value of the pointer is displayed as the address of the data to which the pointer points. This address not necessarily the array element with the lowest address.

TotalView implicitly handles any slicing operations used to set up a pointer, or assumed shape subroutine argument, so that the indices and values which it displays in the variable window for such a variable are the same as you would see in the Fortran code.

For instance, in this code

```
integer, dimension(10), target :: ia
integer, dimension(:), pointer :: ip
do i = i,10
  ia(i) = i
end do
ip => ia(10:1:-2)
```

after diving through the pointer value itself, **ip** displays shown in Figure 62.



```
ia (13168)
 (at 0x00044dfc) Type: integer*4(10)
                 Slice: (:)

Index    Value

(1)      1 (0x00000001)
(2)      2 (0
(3)      3 (0
(4)      4 (0
(5)      5 (0
(6)      6 (0
(7)      7 (0
(8)      8 (0
(9)      9 (0
(10)     10 (
```

```
ip (13168)
> (at 0x00044e20) Type: integer*4(:)
          Actual Type: integer*4(5)
                Slice: (:)

Index    Value

(1)      10 (0x0000000a)
(2)      8 (0x00000008)
(3)      6 (0x00000006)
(4)      4 (0x00000004)
(5)      2 (0x00000002)
```

Figure 62. F90 Pointer Value

This example also shows why the address displayed for the data pane is not that of the base of the array. Since the stride in the array descriptor is negative, succeeding elements of the array are at lower absolute addresses. The address displayed is that of the array element with the lowest index (which may not be the first displayed element if you used a slice to display the array with reversed indices).

Displaying Large Arrays

TotalView can quickly display very large arrays in variable windows. If an array overlaps nonexistent memory, the initial portion of the array is correctly formatted. The array elements that fall within nonexistent memory, have “Bad Address” displayed in the subscript.

Displaying Array Slices

TotalView can display subsections of arrays, which are called *slices*. Every TotalView variable window that displays an array contains an additional **Slice** field. You can edit this field to view subsections of your array. If the array has more than one dimension, then you get the appropriate number of null slices, so, for a C array declared

```
    integer ia[10][20][5]
```

the initial slice will be [:][:][:].

For an F90 deferred shape array declared

```
    integer, dimension (:,:) :: ia
```

the initial slice will be (:,:).

In other words, you get as many colons (**:**) as there are array dimensions. Initially, the field contains either **[:]** for C arrays or **(:)** for Fortran arrays, which displays the entire array.

Slice Descriptions

A slice description consists of the following:

*lower_bound***:***upper_bound***:***stride*

This description specifies that TotalView should display every *stride* element of the array, starting at the *lower_bound* and continuing through the *upper_bound*, inclusive.

For example, if you specified a slice of **[0:9:9]** for a 10-element C array, TotalView displays the first element and last element (the 9th element beyond the lower bound).

TotalView accepts array slices which are the same as those in Fortran 90, so the slice **[lb:ub:stride]** represents the set of values of **i** generated by the **append** statements in the following pseudo-code:

```
i = lb
if (stride > 0)
  while ( i <= ub )
  append i
  i = i + stride
else
  while ( i >= ub )
  append i
  i = i + stride
```

In addition, TotalView accepts a number of extensions to the slices Fortran 90 would accept, since we assume that you want to have some elements in the slice.

Therefore, TotalView will treat a slice

```
[lb : ub : stride]
```

where **stride** < **0** and **ub** > **lb** as though it was intended to be the slice

```
[ub : lb : stride]
```

and will reflect it as such in the slice display.

This extension also means that you can view a dimension with reversed indexing by using the slice

```
[::-1]
```

In Fortran 90, you would have to explicitly give the upper and lower bounds of the array to generate a suitable reverse indexed slice.

In the case where the stride of a slice is **1**, you can specify the slice with just two numbers separated by colons: the lower and upper bounds. For example, to display a slice of **[0:9:1]**, you can specify the following:

```
[0:9]
```

The slice **[0:9]** displays array elements 0 through 9, whereas the slice **[4:6]** displays array elements 4 through 6.

If the stride is **1** and the lower and upper bound are the same number, you can specify the slice with just a single number, which indicates both the lower and upper bound. For example, to display a slice of **[9:9:1]**, you can specify the following:

```
[9]
```

The slice **[9]** displays element 9.

Note: The *lower_bound*, *upper_bound*, and *stride* portions of a slice description must be constant values. Expressions are not supported yet.

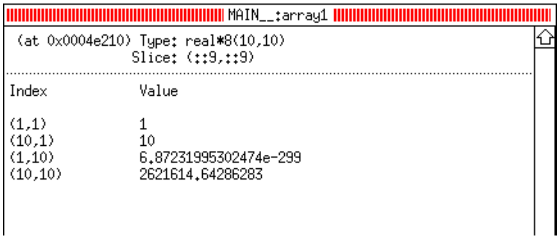
For multidimensional arrays, you can specify a slice for each dimension using the following syntax:

C and C++	**[***slice***][***slice***]**…
Fortran	**(***slice***,***slice***,**…**)**

Strides

You can use the stride of a slice either to skip elements of an array or to invert the order in which elements of an array are displayed.

For example, if you specify a slice of **[::2]** for a C or C++ array (with a default lower bound of 0), TotalView displays only the even elements of the array: 0, 2, 4, and so on. However, if you specify this same slice for a Fortran array (with a default lower bound of 1), TotalView displays only the odd elements of the array: 1, 3, 5, and so on. As an example of skipping elements in a multidimensional array, you can specify a slice of **(::9,::9)** to display the four corners of a 10-element by 10-element Fortran array, as shown in Figure 63.



```
 (at 0x0004e210) Type: real*8(10,10)
                 Slice: (::9,::9)
Index              Value

(1,1)              1
(10,1)             10
(1,10)             6.87231995302474e-299
(10,10)            2621614.64286283
```

Figure 63. Slice Displaying the Four Corners of an Array

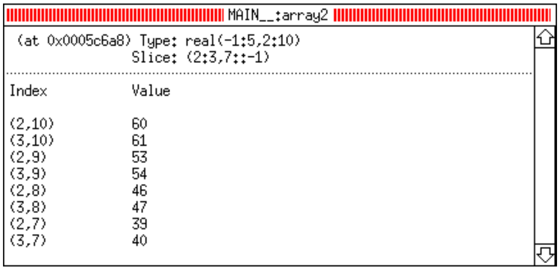
To invert the order in which elements are displayed, you can specify a negative number as the stride of a slice. If you specify a slice of **(::–1)**, TotalView begins with the upper bound of the array and displays the array in inverted order. For example, if you specified this slice of **(::–1)** with a Fortran array of **integer(10)**, TotalView displays the following elements:

```
(10)
(9)
(8)
...
```

You can use a stride to combine inverse order with skipping elements. For example, if you specify a slice of **(::–2)**, TotalView begins with the upper bound of the array and displays every other element until it reaches the lower bound of the array. For example, if you specify this slice of **(::–2)** with a Fortran array of **integer(10)**, TotalView displays the following elements:

```
(10)
(8)
(6)
...
```

You can also combine inverse order and a limited extent to display a small section of a large array. For example, if you specified a slice of **(2:3,7::–1)** with a Fortran array of **real*4(–1:5,2:10)**, Figure 64 shows the elements that are displayed by TotalView:



```
MAIN__:array2
 (at 0x0005c6a8) Type: real(-1:5,2:10)
                 Slice: (2:3,7::-1)

Index            Value

(2,10)           60
(3,10)           61
(2,9)            53
(3,9)            54
(2,8)            46
(3,8)            47
(2,7)            39
(3,7)            40
```

Figure 64. Fortran Array with Inverse Order and Limited Extent

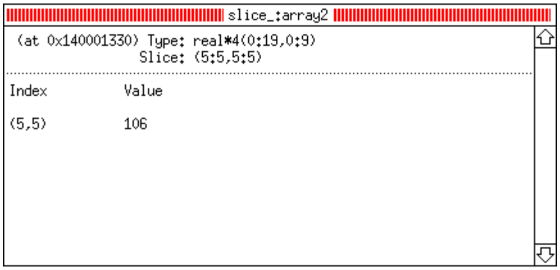
As you can see in the figure, TotalView only shows in rows 2 and 3 of the array, beginning with column 10 and concluding with column 7.

Using Slices in the Variable Command

When you use the **Variable (v)** command to display a variable window, you can include a slice expression as part of the variable name. Specifically, if you include an array name followed by a set of slice descriptions in the variable dialog box, TotalView initializes the slice field in the variable window to the slice descriptions that you specified.

If you include an array name followed by a list of subscripts in the variable dialog box, TotalView interprets the subscripts as a slice description rather than as a request to display an individual value of the array. As a result, you can display different values of the array by changing the slice expression.

For example, suppose that you have a 20-element by 10-element Fortran array named **array2**, and you want to display element **(5,5)**. Using the **Variable (v)** command, you specify **array2(5,5)** in the dialog box, which sets the initial slice to **(5:5,5:5)**, as shown in Figure 65.



```
slice_:array2
 (at 0x140001330) Type: real*4(0:19,0:9)
                   Slice: (5:5,5:5)
Index            Value

(5,5)            106
```

Figure 65. Variable Window for **array2**

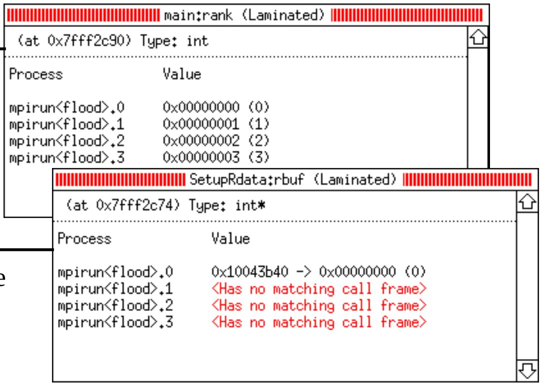
If desired, you can force TotalView to display a single value in a variable window by enclosing the array name and list of subscripts (that is, the information normally included in a slice expression) inside parentheses, such as **(array2(5,5))**. In this case, the variable window just displays the type and value of the element and does not show its array index.

Displaying a Variable in All Processes or Threads

When you debug a parallel program that is running many instances of the same executable, or a multithreaded program, it is often useful to view or update the value of a variable in all of the processes (or threads) at once.

To display the value of a variable in *all* of the processes in a parallel program, first bring up a data pane displaying the value of a variable in one of the processes. Then you can use the **Toggle Laminated Display** (**L**) command from the data pane menu to request that the pane display the value of the variable in all of the processes. To display the value of a variable in all threads within a *single* process, use the **Toggle Thread Laminated Display** (**l**) command. If you decide that you no longer want the pane to be laminated, then you can use the same command to delaminate it, and return it to being a normal data pane.

The data pane switches to "laminated" mode, and displays the value of the variable in each process or thread. Figure 66 shows a display of a simple, scalar variable in each of four processes of an MPI code. In the top window, all of the processes have the variable in a matching stack frame, so the value is displayed for all of them. In the bottom window, a corresponding variable cannot be found, so that information is displayed in the data pane.



Laminated scalar

Laminated scalar with missing call frames in some processes

```
main:rank (Laminated)
(at 0x7fff2c90) Type: int

Process                 Value

mpirun<flood>.0         0x00000000 (0)
mpirun<flood>.1         0x00000001 (1)
mpirun<flood>.2         0x00000002 (2)
mpirun<flood>.3         0x00000003 (3)
```

```
SetupRdata:rbuf (Laminated)
(at 0x7fff2c74) Type: int*

Process                 Value

mpirun<flood>.0         0x10043b40 -> 0x00000000 (0)
mpirun<flood>.1         <Has no matching call frame>
mpirun<flood>.2         <Has no matching call frame>
mpirun<flood>.3         <Has no matching call frame>
```

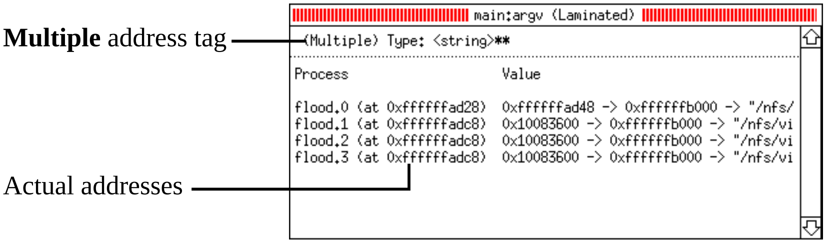
Figure 66. Laminated Scalar Variable

When looking for a matching stack frame to find the correct local variable to display, TotalView matches frames from the outermost frame inwards, and considers calls from different sites to be different, so in code like the following:

```
int recurse (int i, int depth)
{
   if (i == 0)
       return depth;
   if (i&1)
       recurse (i-1, depth+1);
   else
       recurse (i-3, depth+1);
}
```

The two calls to **recurse** generate stack frames that are not considered to match.

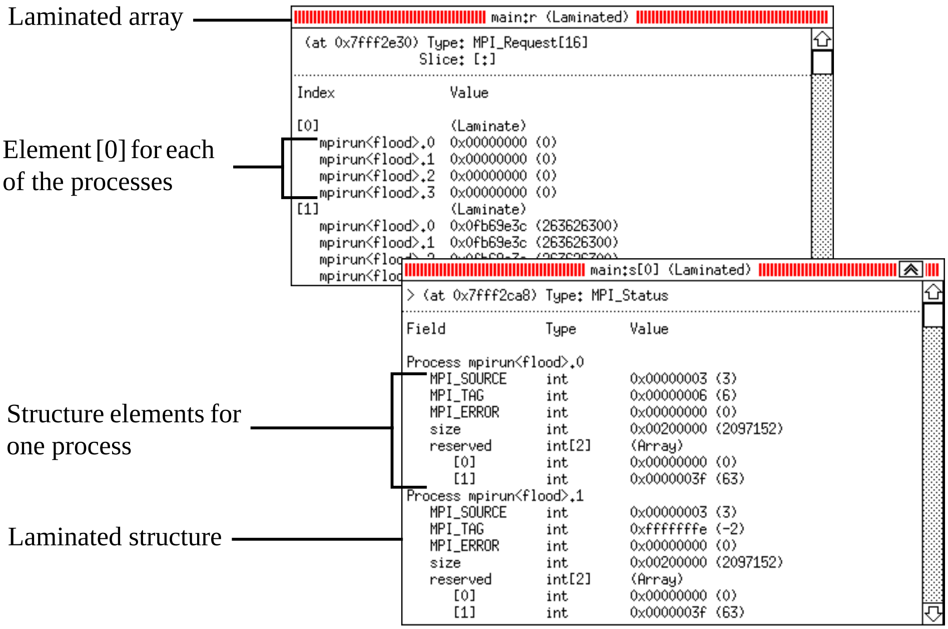
If the variables are at different addresses in the different processes or threads, then the address field at the top of the pane displays **(Multiple)** and the actual addresses are displayed with each data item, as shown in Figure 67.



```
main:argv (Laminated)
(Multiple) Type: <string>**

Process                    Value

flood.0 (at 0xffffffad28)  0xffffffad48 -> 0xffffffb000 -> "/nfs/
flood.1 (at 0xffffffadc8)  0x10083600 -> 0xffffffb000 -> "/nfs/vi
flood.2 (at 0xffffffadc8)  0x10083600 -> 0xffffffb000 -> "/nfs/vi
flood.3 (at 0xffffffadc8)  0x10083600 -> 0xffffffb000 -> "/nfs/vi
```

Figure 67. Laminated Variable at Different Addresses

TotalView also allows you to laminate arrays and structures. When you laminate an array, each element in the array is displayed across all processors. As with a normal data pane, you can use the slice to select elements to be displayed. Structures are displayed to keep the individual structure elements together. Figure 68 shows an example of a laminated array and a laminated structure. You can also laminate an array of structures.



Laminated array

Element [0] for each of the processes

Structure elements for one process

Laminated structure

```
main:r (Laminated)
(at 0x7fff2e30) Type: MPI_Request[16]
            Slice: [:]
Index              Value

[0]                (Laminate)
   mpirun<flood>.0 0x00000000 (0)
   mpirun<flood>.1 0x00000000 (0)
   mpirun<flood>.2 0x00000000 (0)
   mpirun<flood>.3 0x00000000 (0)
[1]                (Laminate)
   mpirun<flood>.0 0x0fb69e3c (263626300)
   mpirun<flood>.1 0x0fb69e3c (263626300)
   mpirun<flo
   mpirun<flo
```

```
main:s[0] (Laminated)
> (at 0x7fff2ca8) Type: MPI_Status
Field              Type       Value

Process mpirun<flood>.0
   MPI_SOURCE      int        0x00000003 (3)
   MPI_TAG         int        0x00000006 (6)
   MPI_ERROR       int        0x00000000 (0)
   size            int        0x00200000 (2097152)
   reserved        int[2]     (Array)
      [0]          int        0x00000000 (0)
      [1]          int        0x0000003f (63)
Process mpirun<flood>.1
   MPI_SOURCE      int        0x00000003 (3)
   MPI_TAG         int        0xfffffffe (-2)
   MPI_ERROR       int        0x00000000 (0)
   size            int        0x00200000 (2097152)
   reserved        int[2]     (Array)
      [0]          int        0x00000000 (0)
      [1]          int        0x0000003f (63)
```

Figure 68. Laminated Array and Structure

Diving in a Laminated Pane

You can dive through pointers in a laminated data pane, and the dive will apply to the appropriate pointer in each process or thread.

Editing a Laminated Variable

If you edit a value in a laminated data pane, then you will be asked whether you want this update to apply to all of the processes or threads or only the one in which you demonstrated the change. Updating a variable in all of the processes is an easy way to turn on a global debug flag, for instance.

Visualizing a Laminated Data Pane

You can export data from a laminated data pane to the visualizer using the Visualize command exactly as for a normal data pane. However the process (or thread) index will form the first axis of the visualization, and therefore you must use one fewer data dimension than you normally would. If you do not want the process/thread axis to be significant to the visualization, then you can simply use a normal data pane, since all of the data must necessarily be in one process.

Visualizing Array Data

The TotalView Visualizer is part of a suite of software development tools for debugging, analyzing and tuning the performance of programs. It works with the TotalView debugger to create graphic images of array data in your programs. This lets you see your data in one glance and quickly find problems with it as you debug your programs.

The visualizer is implemented as a self-contained process. It can be launched directly by TotalView to visualize data as you debug your programs. Alternatively, you can run the visualizer from the command line to visualize data dumped to a file in a previous TotalView session.

You interact with TotalView to choose *what* you want to visualize and *when* the snapshot of your data should be grabbed. You interact with the visualizer to choose *how* you would like your data to be displayed.

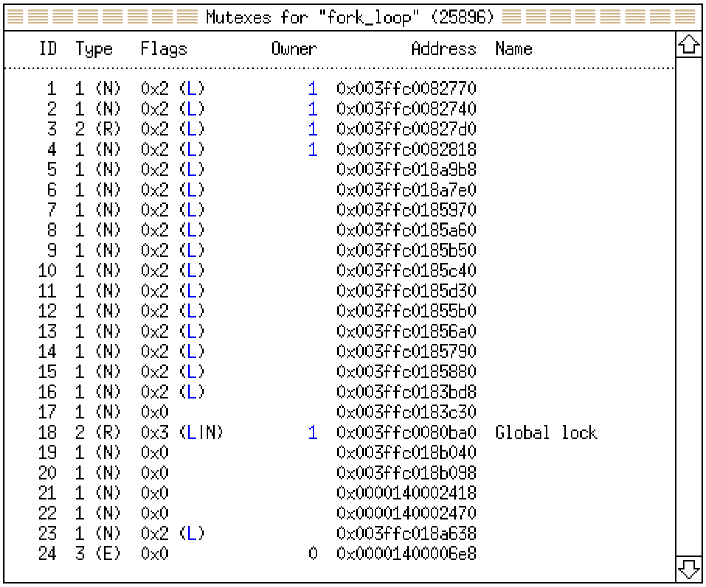
For information about running the TotalView Visualizer, see Chapter 9, “Visualizing Data,” on page 231.

Displaying Mutex Information

A mutex is a mutual exclusion object that allows multiple threads to synchronize access to shared resources. A mutex has two states: locked and unlocked. Once a mutex has been locked by a thread, other threads attempting to lock it will block. When the locking thread unlocks (releases) the mutex, one of the blocked threads will acquire (lock) it and proceed.

Note: The Mutex Information window is supported only on Digital UNIX.

The mutex information window contains a list of all mutual exclusions (mutexes) known in this process. To get a mutex window click on the **Mutex Info Window** command from the **Process State Info** submenu in the process window. See Figure 69.



```
Mutexes for "fork_loop" (25896)

  ID  Type   Flags         Owner          Address  Name
...........................................................
   1  1 (N)  0x2 (L)           1  0x003ffc0082770
   2  1 (N)  0x2 (L)           1  0x003ffc0082740
   3  2 (R)  0x2 (L)           1  0x003ffc00827d0
   4  1 (N)  0x2 (L)           1  0x003ffc0082818
   5  1 (N)  0x2 (L)              0x003ffc018a9b8
   6  1 (N)  0x2 (L)              0x003ffc018a7e0
   7  1 (N)  0x2 (L)              0x003ffc0185970
   8  1 (N)  0x2 (L)              0x003ffc0185a60
   9  1 (N)  0x2 (L)              0x003ffc0185b50
  10  1 (N)  0x2 (L)              0x003ffc0185c40
  11  1 (N)  0x2 (L)              0x003ffc0185d30
  12  1 (N)  0x2 (L)              0x003ffc01855b0
  13  1 (N)  0x2 (L)              0x003ffc01856a0
  14  1 (N)  0x2 (L)              0x003ffc0185790
  15  1 (N)  0x2 (L)              0x003ffc0185880
  16  1 (N)  0x2 (L)              0x003ffc0183bd8
  17  1 (N)  0x0                  0x003ffc0183c30
  18  2 (R)  0x3 (L|N)         1  0x003ffc0080ba0  Global lock
  19  1 (N)  0x0                  0x003ffc018b040
  20  1 (N)  0x0                  0x003ffc018b098
  21  1 (N)  0x0                  0x0000140002418
  22  1 (N)  0x0                  0x0000140002470
  23  1 (N)  0x2 (L)              0x003ffc018a638
  24  3 (E)  0x0               0  0x00001400006e8
```

Figure 69. Mutex Info Window

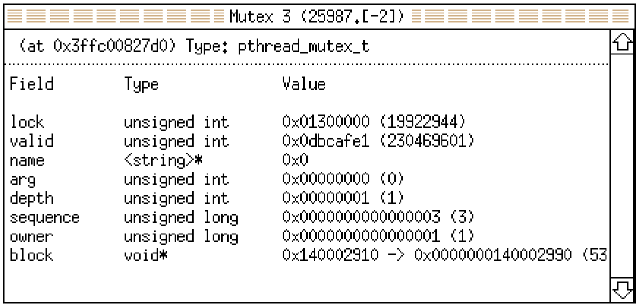
For each mutex, TotalView displays the following information:

- ID. This is the sequence number assigned to this mutex by the threads package. Diving into this field opens a data window containing a view of the actual mutex data.
- Type. The type contains the raw mutex type number, along with a single-character abbreviation of the type name. The following mutex types are known to TotalView:
  - (N) A normal mutex.
  - (R) A recursive mutex.
  - (E) An error-check mutex. Error-check mutexes contain additional information for use in debugging, such as the thread ID of the locker. During program development, you should probably use error-check mutexes in place of normal mutexes, and only switch to the simpler version when performance becomes an issue. The type of the mutex can be set using the **pthread_mutexattr_settype_np()** call on the attribute object before the mutex is initialized.
- Flags. Flags are a raw hex string containing the current mutex flags, along with a text summary showing one-character abbreviations for each flag which is set. The following mutex flags are known to TotalView:
  - 0x8 (M) Metered. The mutex contains metering information.
  - 0x4 (W) Waiters. One or more threads are waiting for this mutex. By default, waiters are shown in red; their color is the same as the thread *error* state flag color.
  - 0x2 (L) Locked. The mutex is locked. By default, locked mutexes are shown in blue; their color is the same as the thread *stopped* state flag color.
  - 0x1 (N) Name. This mutex has a name.
- Owner (Error-check mutexes *only)*. If the mutex is locked, as indicated by the L flag, this field displays the system **tid** of the locking thread. Diving or selecting on this number causes TotalView to display the process window for the locking thread. TotalView displays the same window if you dive or select the thread's entry in the root window.

If threads are waiting for this mutex, their system **tids** will be shown in the owner field, with one thread ID displayed for each line in the window. You can open a process window for these waiting threads by diving or selecting on its number.

Note: TotalView may not be able to obtain this information, in which case it will not show blocked-thread lines.

- Address. The address of the mutex in memory. You can open a data window containing a view of the actual mutex data by diving on this field. See Figure 70.
- Name. If the mutex has a name, it is shown here.



```
Mutex 3 (25987.[-2])
 (at 0x3ffc00827d0) Type: pthread_mutex_t

Field        Type              Value

lock         unsigned int      0x01300000 (19922944)
valid        unsigned int      0x0dbcafe1 (230469601)
name         <string>*         0x0
arg          unsigned int      0x00000000 (0)
depth        unsigned int      0x00000001 (1)
sequence     unsigned long     0x0000000000000003 (3)
owner        unsigned long     0x0000000000000001 (1)
block        void*             0x140002910 -> 0x0000000140002990 (53
```

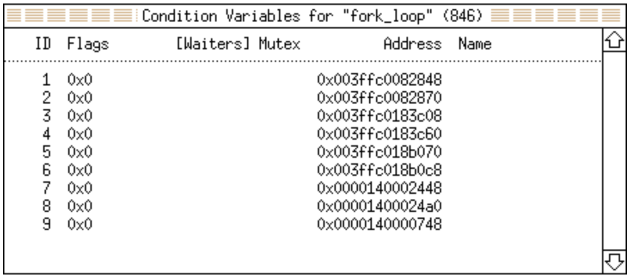
Figure 70. Mutex Data Window

Displaying Condition Variable Information

The window that displays the condition variables lists all the condition variables known in this process.

Note: The Condition Variables window is supported only on Digital UNIX.

To get a Condition Variables window, click on the **Condition Variable Info Window** command in the **Process State Info** submenu of the process window. See Figure 71.



```
Condition Variables for "fork_loop" (846)
   ID  Flags        [Waiters] Mutex         Address  Name

    1  0x0                            0x003ffc0082848
    2  0x0                            0x003ffc0082870
    3  0x0                            0x003ffc0183c08
    4  0x0                            0x003ffc0183c60
    5  0x0                            0x003ffc018b070
    6  0x0                            0x003ffc018b0c8
    7  0x0                            0x000014000244 8
    8  0x0                            0x00001400024a0
    9  0x0                            0x0000140000748
```

Figure 71. Condition Variable Window

For each condition variable, TotalView displays the following information:

- ID. The ID is the sequence number assigned to this condition variable by the threads package. Diving into this field opens a data window containing a view of the actual condition variable data.
- Flags. Flags are a raw hex string containing the current condition variable flags, with a text summary showing one-character abbreviations for each flag which is set. The following flags are known to TotalView:
  - 0x8 (M)

    Metered. This condition variable contains metering information.

  - 0x4 (W)

    Waiters. One or more threads are waiting for this condition variable. By default, this is shown in red; its color is the same as the thread *error* state flag color.

  - 0x2 (P)

    Pending. A wakeup is pending for this condition variable. By default, this is shown in blue; its color is the same as the thread *stopped* state flag color.

  - 0x1 (N)

    Name. The condition variable has a name.

- Waiters. If threads are waiting for this condition variable, their system **tids** will be shown in the **Waiters** field, one thread for each line, on the lines following the condition variable. Diving or selecting entries in the list of waiting threads will open windows for them.

Note: TotalView may not be able to obtain this information, in which case no waiting threads will be shown.

- Mutex. This field has the ID of the mutex used to guard the condition variable. Diving into this field opens a data window containing a view of the actual guard mutex data, if the ID can be translated to an address. For the translation to be possible, the guard mutex must be correctly initialized. That can be done statically or by using an attributes object. See the mutex and condition variable man pages for more information.
- Address. This field contains the address of the condition variable in memory. Diving into the address field opens a data window containing a view of the actual condition variable data.
- Name. If the condition variable has a name, it will be shown here.

CHAPTER 8: Setting Action Points

This chapter explains how to use action points. TotalView supports three kinds of action points: breakpoints, process barrier breakpoints, and evaluation points. A breakpoint stops execution of processes and threads that reach it. A process barrier breakpoint holds each process that reaches it until all processes from the group reach it. An evaluation point causes a code fragment to execute when it is reached.

In this chapter, you'll learn how to:

- Set breakpoints
- Set evaluation points
- Set conditional breakpoints
- Patch programs
- Set process barrier breakpoints
- Choose between interpreting and compiling expressions
- Control action points
- Save action points in a file
- Evaluate expressions
- Write code fragments
- Write assembler code (Alpha Digital UNIX and AIX systems only)

Action Points

Actions points allow you to specify an action to be performed when a thread or process reaches a source line or machine instruction in your program. TotalView support the following types of action points:

- Breakpoints

  Breakpoints are the simplest type of action point. When a thread or process encounters a breakpoint during execution, it stops at the breakpoint along with the other threads in the process. You can also arrange for other related processes to stop when the breakpoint is hit.

- Process barrier breakpoints

  Process barrier breakpoints are similar to simple breakpoints, but they useful for synchronizing a group of processes in a multiprocess program. Process barrier breakpoints work together with the TotalView hold and release process feature.

- Evaluation points

  Evaluation points allow you to specify a code fragment to be executed when the thread or process reaches the evaluation point. Evaluation points can be used in several different ways, including conditional breakpoints, thread - specific breakpoints, countdown breakpoints, and patching code fragments into and out of your program.

All of the different type of action points share some common properties:

- Action points can be set at a source line or machine instruction.
- Action points can be enabled or disabled independently, which allows you to retain the action point definition, but remove it from your program.
- Action points can be shared across multiple processes, or set in individual processes.
- Action points are apply to the process, so in a multithreaded process, it applies to all of the threads.
- Action points are assigned unique action point ID numbers. They appear in several places, including: the root window, the action points pane of the process window, and the action points dialog box.

Each type of action point has its own symbol associated with it. Figure 72 shows examples of **STOP** (breakpoint), **BARR** (process barrier breakpoint), and **EVAL** (evaluation point) symbols, both enabled and disabled, and **ASM** (assembler-level action point) symbol.

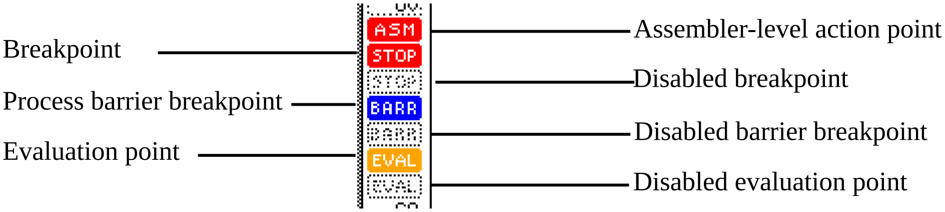

Breakpoint
Process barrier breakpoint
Evaluation point
ASM
STOP
STOP
BARR
BARR
EVAL
EVAL
Assembler-level action point
Disabled breakpoint
Disabled barrier breakpoint
Disabled evaluation point

Figure 72. Action Point Symbols

The ASM symbol indicates that there are one or more assembler-level action points associated with the source line.

The following sections describe the different types of actions points in more detail.

Setting Breakpoints

The TotalView debugger offers several options for setting breakpoints. You can set source-level breakpoints, machine-level breakpoints, and breakpoints that are shared among all processes in multiprocess programs. You can also control whether or not TotalView stops all processes in the program group when a single member reaches a breakpoint.

Note: Breakpoints apply to the entire process, not just to a single thread. Any thread executing in the process could reach the breakpoint, thus causing it to stop.

Setting Source-Level Breakpoints

There are several ways to set source-level breakpoints in TotalView. Typically, you set and clear breakpoints before you start a process. To set a source-level breakpoint, select a boxed line number in the tag field of the process window. A boxed line number indicates that the line generates executable code. A STOP sign, shown in Figure 73, indicates that the breakpoint occurs *before* the source statement is executed.

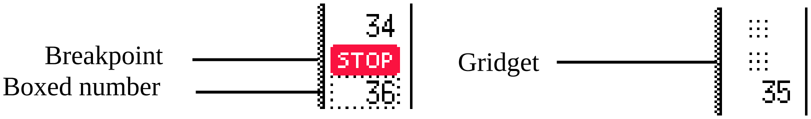
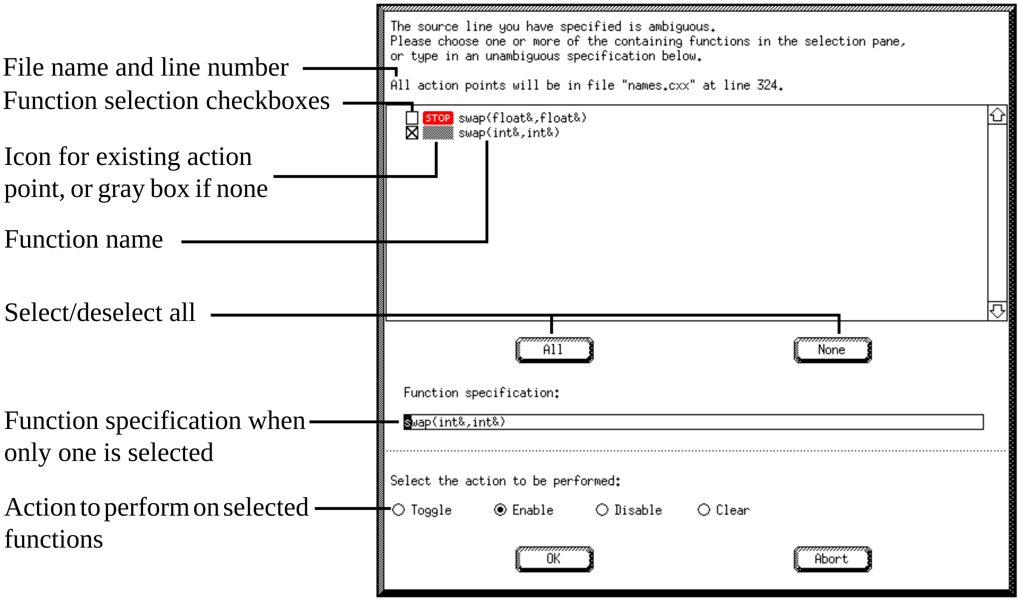


Figure 73. Breakpoint Symbol

You can also set a breakpoint while a process is running by selecting a boxed line number in the tag field of the process window. If you set the breakpoint while the process is running, TotalView stops the process temporarily to insert the breakpoint and then continues running it.

Selecting Ambiguous Source Lines

If you are using C++ templates, it is possible that a single source line could generate multiple function instances. If you attempt to set a source-level breakpoint by selecting a line number in a function template, and that line number has more than one instantiation, TotalView will prompt you with an ambiguous source line selection dialog box, as shown in Figure 74.



The source line you have specified is ambiguous.
Please choose one or more of the containing functions in the selection pane,
or type in an unambiguous specification below.

All action points will be in file "names.cxx" at line 324.

□ STOP swap(float&,float&)
☒ ▨ swap(int&,int&)

Function specification:

swap(int&,int&)

Select the action to be performed:

○ Toggle ● Enable ○ Disable ○ Clear

File name and line number
Function selection checkboxes
Icon for existing action point, or gray box if none
Function name
Select/deselect all
Function specification when only one is selected
Action to perform on selected functions
All
None
OK
Abort

Figure 74. Ambiguous Source Line Selection Dialog Box

Perform the following steps to resolve the ambiguity.

1. Select the set of functions to operate on by selecting:
   - the **All** button to select all of the functions
   - the **None** button to deselect all the functions
   - individual checkboxes to select and deselect a function

Note: The function specification is automatically set to the currently selected function when exactly one box is checked. Selecting additional checkboxes clears this field.

---

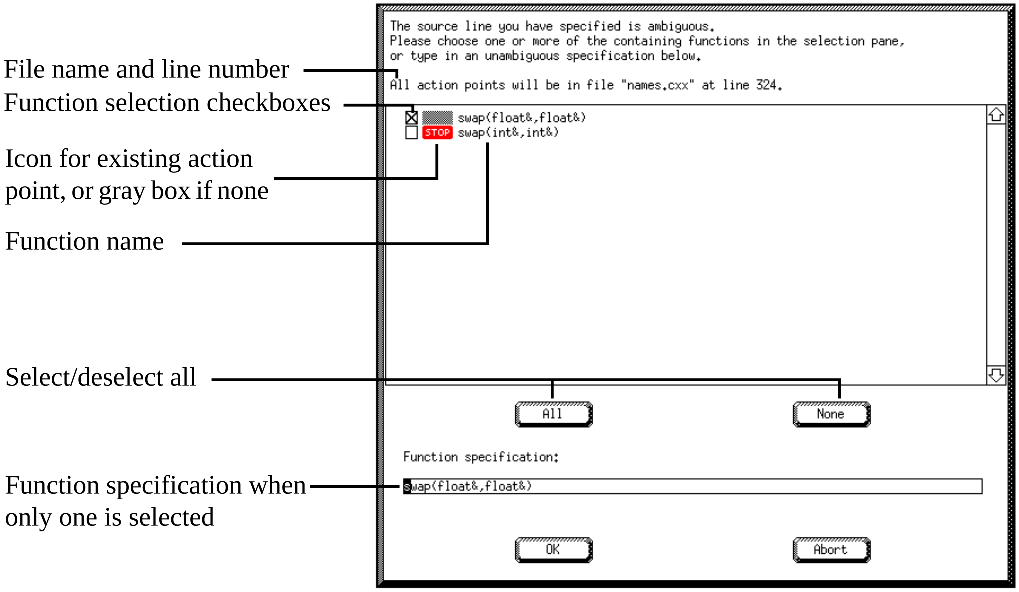
2. Select the action to perform on the set selected functions by selecting the:
   - **Toggle** radio button to toggle the state of the action points
   - **Enable** radio button to enable the action points, or create breakpoints or process barrier breakpoints for any that did not already exist
   - **Disable** radio button to disable the action point

- **Clear** radio button to delete default breakpoints or process barrier breakpoints, and disable others

3. Select the OK button, or press Return to perform the action. If you hold down the Shift key, the action performed will be for process barrier breakpoints.

Diving into Ambiguous Source Lines

Similar to selecting an ambiguous source line, if you dive on an ambiguous source line, TotalView displays the ambiguous source line dive dialog box, shown in Figure 75, before posting the action point options dialog box.



```
The source line you have specified is ambiguous.
Please choose one or more of the containing functions in the selection pane,
or type in an unambiguous specification below.

All action points will be in file "names.cxx" at line 324.
```

☒ ▨ swap(float&,float&)
□ STOP swap(int&,int&)

Function specification:

swap(float&,float&)

File name and line number
Function selection checkboxes
Icon for existing action point, or gray box if none
Function name
Select/deselect all
Function specification when only one is selected
All
None
OK
Abort

Figure 75. Ambiguous Source Line Dive Dialog Box

Perform the following steps to resolve the ambiguity:

1. Select the set of functions to operate on by selecting:

   - the **All** button to select all of the functions
   - the **None** button to deselect all the functions

- individual checkboxes to select and deselect a function

Note: The function specification is automatically set to the currently selected function when exactly one box is checked. Selecting additional checkboxes clears this field.

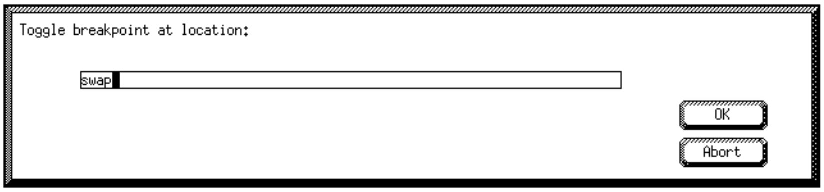
2. Select the OK button, or press Return to display the action point options dialog box. Any changes made in the action point options dialog box will be applied to the selected functions.

As with other action point function menus, you are allowed to specify multiple functions. However, if you do, the source lines that are referenced must all contain no action points, or contain action points of the same type. The reason for this is that once the action points are selected, a standard action point options dialog box appears, and the selections you make in this dialog box apply to *all* the action points that you have selected.

Toggling Breakpoints at Locations

You can toggle a breakpoint at a specific function or source line number without having to first find the function or source line in the source pane. To set a breakpoint this way:

1. Issue the **Breakpoint at Location (^B)** command in the **STOP/BARR/EVAL/GIST** submenu of the process window. The toggle breakpoint dialog box appear as shown in Figure 76



Toggle breakpoint at location:

swap

Figure 76. Toggle Breakpoint at Location Dialog Box

2. Enter the name of the function or a source line number.

   If you enter a the name of a function, the breakpoint will be toggled at the first executable source line in the function you specified. If you enter a source line number, the breakpoint will be toggled at the source line in the current source file.

3. Select the OK button, or press Return. If you hold down the Shift key, this command will perform toggle a process barrier breakpoint at this location.

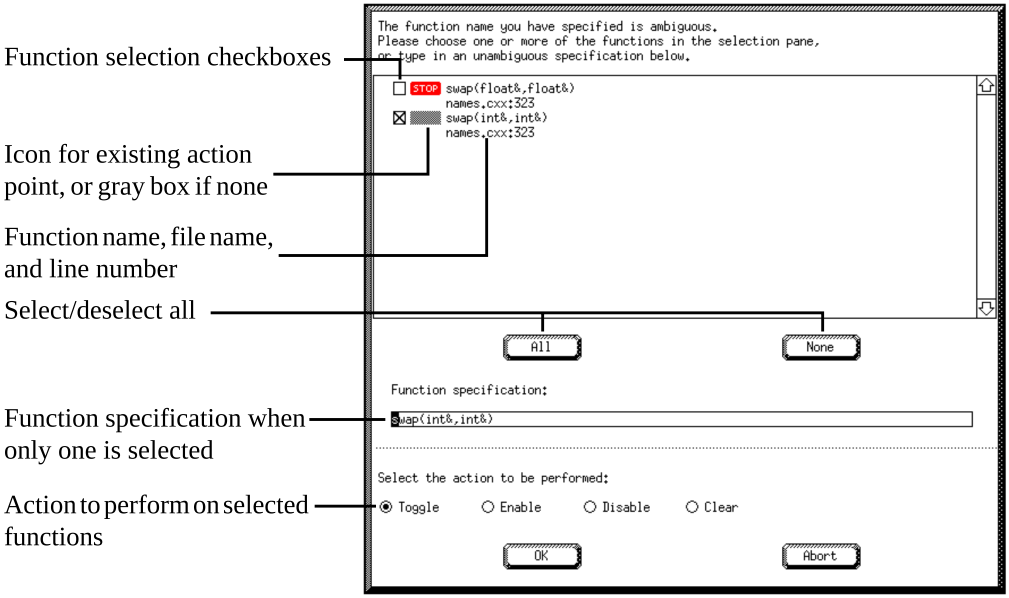
The behavior of the **Breakpoint at Location (^B)** command depends on whether there is already an action point at the selected location, and whether hold down the Shift key when you select OK or press Return, as described in Table 13.

Table 13. Breakpoint at Location Actions

Location Content	**OK Action**	**Shift-OK Action**
Empty	Create **STOP**	Create **BARR**
STOP	Delete/disable **STOP**	Convert to **BARR**
BARR	Delete/disable **BARR**	Convert to **STOP**
EVAL	Disable **EVAL**	Disable **EVAL**

Toggling Breakpoints at Ambiguous Locations

If you give the **Breakpoint at Location (^B)** command an ambiguous function name, TotalView prompts you with an ambiguous function dialog box, as shown in Figure 77.


Function selection checkboxes
Icon for existing action point, or gray box if none
Function name, file name, and line number
Select/deselect all
Function specification when only one is selected
Action to perform on selected functions
All
None
OK
Abort

```
The function name you have specified is ambiguous.
Please choose one or more of the functions in the selection pane,
or type in an unambiguous specification below.
```

□ STOP swap(float&,float&)
names.cxx:323

☒ ▨ swap(int&,int&)
names.cxx:323

Function specification: swap(int&,int&)

Select the action to be performed:

● Toggle ○ Enable ○ Disable ○ Clear

Figure 77. Ambiguous Function Name Dialog Box

Perform the following steps to resolve the ambiguity:

1. Select the set of functions to operate on by selecting:
   - the **All** button to select all of the functions
   - the **None** button to deselect all the functions
   - individual checkboxes to select and deselect a function

Note: The function specification is automatically set to the currently selected function when exactly one box is checked. Selecting additional checkboxes clears this field.

---

2. Select the action to perform on the set selected functions by selecting the:
   - **Toggle** radio button to toggle the state of the action points
   - **Enable** radio button to enable the action points, or create breakpoints or process barrier breakpoints for any that did not already exist
   - **Disable** radio button to disable the action point
   - **Clear** radio button to delete default breakpoints or process barrier breakpoints, and disable others
3. Select the OK button, or press Return. If you hold down the Shift key, the action performed will be for process barrier breakpoints.

Setting Machine-Level Breakpoints

To set a machine-level breakpoint, you must first display assembler code or source interleaved with assembler. (Refer to "Examining Source and Assembler Code" on page 120 for information.)

Then you select the tag field that is opposite the appropriate instruction. The tag field must contain a gridget, which indicates the line is the beginning of a machine instruction. Since the instruction sets on some platforms support variable-length instructions, you may see multiple lines associated with a single gridget. The stop sign appears, indicating that the breakpoint occurs *before* the instruction is executed.

Note: When the source pane displays source interleaved with assembler, source statements are treated as comments. You can set breakpoints on instructions, not source statements. If you set a breakpoint on the first instruction after a source statement, however, you actually create a source-level breakpoint.

If you set machine-level breakpoints on one or more instructions that are part of a single source line and then display source code in the source pane, TotalView displays an ASM sign (see Figure 72) on the line number. To see the specific breakpoints, you must display assembler or assembler interleaved with source code.

After you set all desired breakpoints, you can start the process. When a process reaches a breakpoint, TotalView does the following:

- Suspends the process
- Displays the PC symbol over the stop sign to indicate the PC currently points to the breakpoint
- Displays "at breakpoint" in the title bar of the process window and other windows
- Updates the stack trace panes, stack frame panes, and variable windows.

Thread-Specific Breakpoints

TotalView implements thread-specific breakpoints through evaluation points in the TotalView expression system. The expression system has several intrinsic variables that allow a thread to retrieve its thread ID. For example, the following shows how to set a breakpoint that stops the process only when thread 3 executes the evaluation point:

```
/* Stop when thread 3 evaluates this expression. */
if ($tid == 3) $stop;
```

Breakpoints for Multiple Processes

In multiprocess programs, you can set breakpoints in the parent process and child processes before you start the program and at any time during its execution. To do this, you use the action point options dialog box, as shown in Figure 78. This dialog box provides three checkboxes for process groups:

- Stop All Related Processes when Breakpoint Reached

  If selected, stops all members of the program group when the breakpoint is reached. Otherwise, only the process that reaches the breakpoint stops.

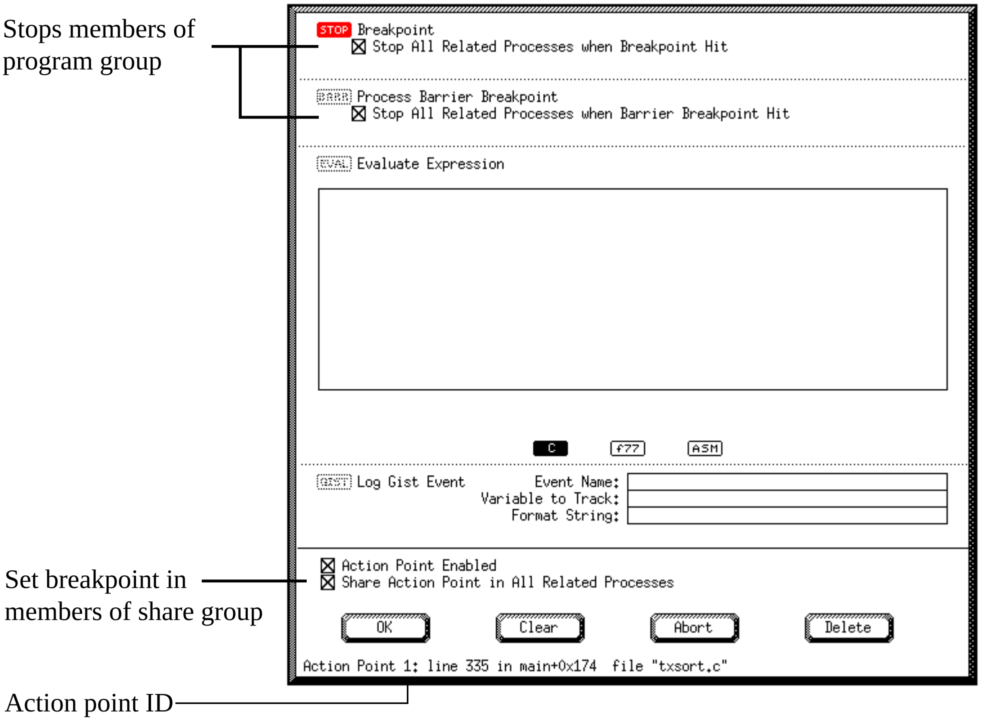
- Stop All Related Processes when Barrier Breakpoint Hit

  If selected, stops all members of the program group when the barrier breakpoint is reached. Otherwise, only the process that reaches the barrier breakpoint stops.

- Share Action Point in All Related Processes

  If selected, enables and disables the breakpoint in all members of the share group at the same time. Otherwise, you enable and disable the breakpoint in each share group member individually.

You can control the default setting of these checkboxes using X resources or command line options. See Figure 78.



Stops members of program group

Set breakpoint in members of share group

Action point ID

STOP Breakpoint
☒ Stop All Related Processes when Breakpoint Hit

BARR Process Barrier Breakpoint
☒ Stop All Related Processes when Barrier Breakpoint Hit

EVAL Evaluate Expression

GIST Log Gist Event
Event Name:
Variable to Track:
Format String:

☒ Action Point Enabled
☒ Share Action Point in All Related Processes

Action Point 1: line 335 in main+0x174 file "txsort.c"

Figure 78. Action Point Options Dialog Box

Refer to “totalview*stopAllRelatedProcessesWhenBreakpointHit: {true | false}” on page 280, “totalview*processBarrierStopAllRelatedProcessesWhenBreakpointHit: {true | false}” on page 276, and “totalview*shareActionPointInAllRelatedProcesses: {true | false}” on page 278 and “TotalView Command Syntax” on page 287.

In addition to the controls in the action point options dialog, you can write an expression in the expression box to control the behavior of program group members and share group members. Refer to “Writing Code Fragments” on page 218 for more information.

Breakpoint for Programs that fork()/execve()

You must link with the dbfork library to debug programs that call **fork()** and **execve()**. See “Compiling Programs” on page 16.

Processes That Call fork()

By default, breakpoints are shared by all processes in the share group, and when any process reaches the breakpoint, TotalView stops all processes in the program group.

To override these defaults:

1. Dive into the tag field to display the action point options dialog box.
2. Deselect these checkboxes: **Stop All Related Processes when Breakpoint Hit** and **Share Action Point in All Related Processes**.
3. Select the **OK** button.

Processes That Call execve()

Breakpoints that are shared by a parent and children with the same executable do not apply to children with different executables. To set the breakpoints for children that call **execve()**:

1. Set the breakpoints and breakpoint options desired in the parent and the children that do *not* call **execve()**.
2. Start the multiprocess program by displaying the **Go/Halt/Step/Next/Hold** submenu and selecting the **Go Group (G)** command. When the *first* child calls **execve()**, a dialog box appears with the following message:

   ```
   Process name has called exec (name),
   Do you wish to stop it before it enters MAIN?
   ```
3. Answer **Yes**. TotalView opens a process window for the process. (If you answer **No**, the program executes without allowing you to set breakpoints.)
4. Set the breakpoints desired for the process. Once you set the breakpoints for the first child that uses this executable, the debugger does not prompt you when other children call **execve()** to use this executable. Therefore, if you *don’t* want to share the breakpoints among other children using the same executable, dive into the breakpoints, and set the breakpoint options appropriately.
5. Select the **Go Group (G)** command from the **Go/Halt/Step/Next/Hold** menu to resume execution.

Example: Multiprocess Breakpoint

The following example program illustrates the different points at which you can set breakpoints in multiprocess programs:

```
1 pid = fork();
2 if (pid == -1)
3          error ("fork failed");
4 else if (pid == 0)
5          children_play();
6 else
7          parents_work();
```

Table 14 shows the results of setting a breakpoint on different lines of the example.

Table 14. Setting Breakpoints in Multiprocess Programs

Line Number	Result
1	Stops the parent process before it forks.
2	Stops both the parent and child processes (if the child process was successfully created).
3	Stops the parent process if **fork()** failed.
5	Stops the child process.
7	Stops the parent process.

Process Barrier Breakpoints

A process barrier breakpoint (process barrier point) is a just like a simple breakpoint, but it holds processes that reach the process barrier point. TotalView holds each process until all the processes in the group reach the same process barrier point. When the last process reaches the same barrier point, all processes in the group are released.

Process Barrier Breakpoint States

Processes at a process barrier point are held or stopped, as follows:

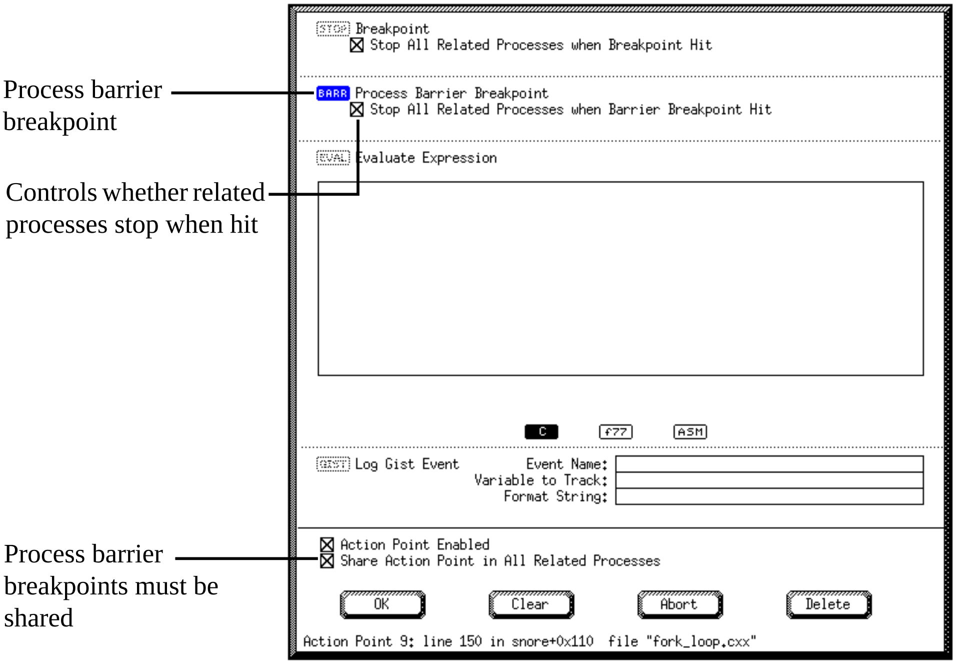
- **Held.** A process that is held cannot resume execution until all the processes in its group are at the process barrier point, or until you manually release it. When held, the various "Go" and "Single-step" commands from the **Go/Halt/Stop/Next/Hold** menu have no effect on held process.
- **Stopped.** When all the processes in the group reach the process barrier point, TotalView automatically releases them. They remain stopped at the barrier point until you take action on them.

You can manually release held processes by choosing **Hold/Release Process (w)** or **Release Group** command from the **Go/Halt/Stop/Next/Hold** menu. When you manually release a process, the "Go" and "Single-step" commands become available again. You can use the **Hold/Release Process** (**w**) command again to toggle the hold state of the process. See "Holding and Releasing Processes" on page 128 for more information.

Setting a Process Barrier Breakpoint

You can set a process barrier breakpoint with the mouse or from the action points dialog box.

- To set a process barrier breakpoint with the mouse, move the mouse to the line number in the process window where you want to set the process barrier point. Then press **Shift-Select**. A **BARR** sign appears. See Figure 72 on page 189.
- To set a process barrier breakpoint from the action point options dialog box, dive on the line where you want to set the process barrier point. In the action points options dialog box, click on the **BARR** sign, then click on **OK**. See Figure 79.



Process barrier breakpoint

Controls whether related processes stop when hit

Process barrier breakpoints must be shared

STOP Breakpoint
☒ Stop All Related Processes when Breakpoint Hit

BARR Process Barrier Breakpoint
☒ Stop All Related Processes when Barrier Breakpoint Hit

EVAL Evaluate Expression

GIST Log Gist Event
Event Name:
Variable to Track:
Format String:

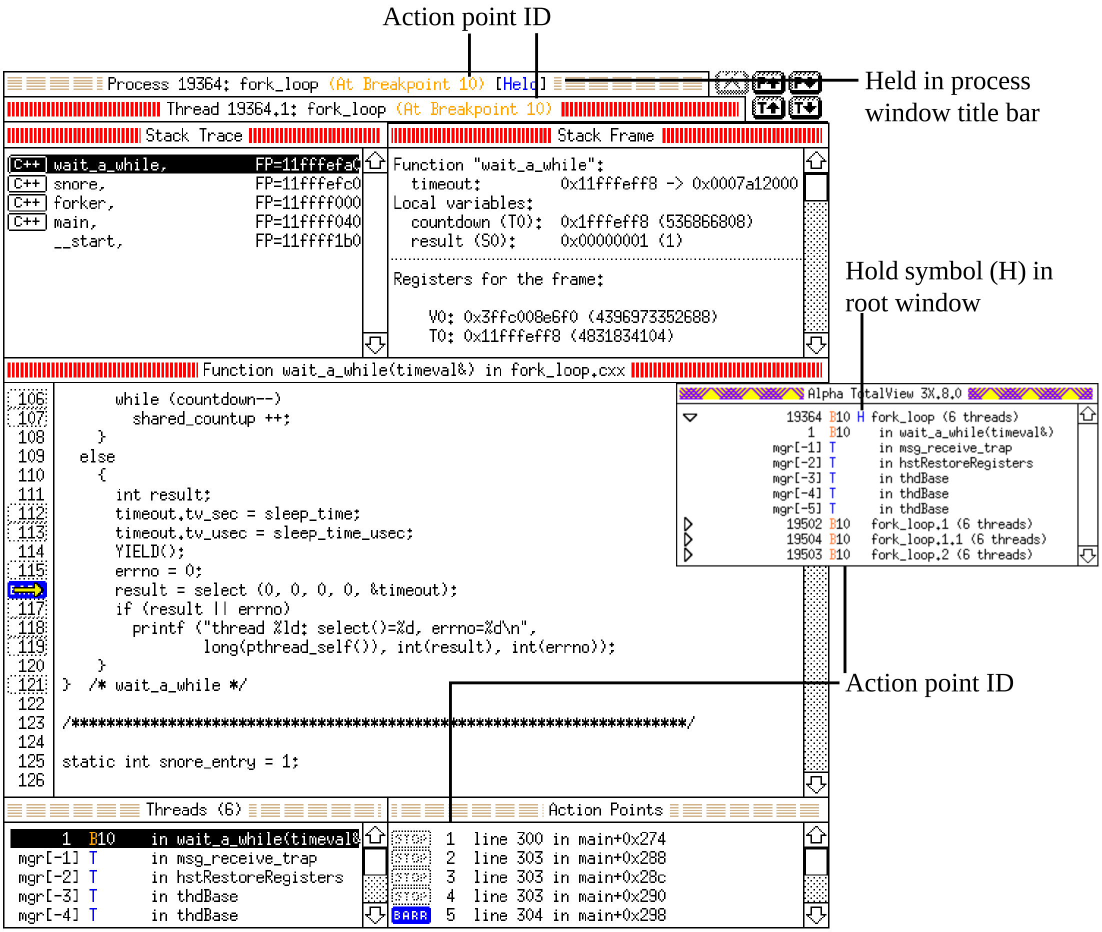
☒ Action Point Enabled
☒ Share Action Point in All Related Processes

Action Point 9: line 150 in snore+0x110 file "fork_loop.cxx"

Figure 79. Action Point Options Dialog Box

When you set a process barrier point, TotalView places it in all the processes in the group. TotalView insists that you create barrier breakpoints that are active in the entire share group.

If you run one of the processes in the group and it hits the process barrier point, you will see an **H** next to the process name in the root window and the word [**Held**] in the process title bar in the main process window. Process barrier points are always shared. See Figure 80.



Action point ID

Held in process window title bar

Hold symbol (H) in root window

Action point ID

```
Process 19364: fork_loop (At Breakpoint 10) [Held]
Thread 19364.1: fork_loop (At Breakpoint 10)

Stack Trace
C++ wait_a_while,          FP=11fffefa0
C++ snore,                 FP=11fffefc0
C++ forker,                FP=11ffff000
C++ main,                  FP=11ffff040
    __start,               FP=11ffff1b0

Stack Frame
Function "wait_a_while":
  timeout:         0x11fffeff8 -> 0x0007a12000
Local variables:
  countdown (T0):  0x1fffeff8 (536866808)
  result (S0):     0x00000001 (1)

Registers for the frame:

    V0: 0x3ffc008e6f0 (4396973352688)
    T0: 0x11fffeff8 (4831834104)

Function wait_a_while(timeval&) in fork_loop.cxx
      while (countdown--)
        shared_countup ++;
    }
  else
    {
      int result;
      timeout.tv_sec = sleep_time;
      timeout.tv_usec = sleep_time_usec;
      YIELD();
      errno = 0;
      result = select (0, 0, 0, 0, &timeout);
      if (result || errno)
        printf ("thread %ld: select()=%d, errno=%d\n",
                long(pthread_self()), int(result), int(errno));
    }
} /* wait_a_while */

/**********************************************************************/

static int snore_entry = 1;

Alpha TotalView 3X.8.0
            19364 B10 H fork_loop (6 threads)
                1 B10      in wait_a_while(timeval&)
          mgr[-1] T        in msg_receive_trap
          mgr[-2] T        in hstRestoreRegisters
          mgr[-3] T        in thdBase
          mgr[-4] T        in thdBase
          mgr[-5] T        in thdBase
            19502 B10   fork_loop.1 (6 threads)
            19504 B10   fork_loop.1.1 (6 threads)
            19503 B10   fork_loop.2 (6 threads)

Threads (6)
        1  B10      in wait_a_while(timeval&
  mgr[-1]  T        in msg_receive_trap
  mgr[-2]  T        in hstRestoreRegisters
  mgr[-3]  T        in thdBase
  mgr[-4]  T        in thdBase

Action Points
STOP  1  line 300 in main+0x274
STOP  2  line 303 in main+0x288
STOP  3  line 303 in main+0x28c
STOP  4  line 303 in main+0x290
BARR  5  line 304 in main+0x298
```

Figure 80. Process Barrier Breakpoint in Process and Root Windows

Releasing Processes from Process Barrier Points

TotalView automatically releases processes from a process barrier point in the following situations:

- A process hits that process barrier point after all other processes in the group are already held at it

- You can create a new process barrier point, and every process in the group is already stopped at the location of the new barrier. Normally, when you create a new process barrier point TotalView holds any process which is stopped at the barrier's location. However, rather than holding *all* the processes in this case, TotalView leaves them all not held.

Toggling Between a Breakpoint and a Process Barrier Point

You can convert an ordinary breakpoint to a process barrier point by moving the cursor to the breakpoint and clicking **Shift-Select**. To convert a process barrier point back to an ordinary breakpoint, move the cursor to the process barrier breakpoint and use **Shift-Select**.

Plain select clears barrier points, just as it clears breakpoints.

Note: **Shift-Select** on an **Eval** point does *not* convert it to a process barrier point.

Deleting a Process Barrier Point

You can delete a process barrier point from the action points dialog box or from the process window.

- If the process barrier point was created with default settings, simply select the **BARR** symbol in the source pane of the process window to delete it. Otherwise, if some options have been set to non-default values, when you select it TotalView just disables it. If you want to re-enable it with the same options you had previously, select it again.
- To delete a process barrier point, or other action point, which has non-default options, dive on the action point symbol in the source pane of the process window to display the action points dialog box. In the dialog box, click **Delete**.

Changes when Setting and Clearing a Barrier Point

Setting a process barrier point at the current PC for a *stopped* process holds the process there, unless all other processes in its group are at that same PC. If they are, TotalView does not hold them. They are the same as if they were stopped at an ordinary breakpoint.

All processes which are held and which have threads at the process barrier point are released when you clear the barrier point. They remain stopped, but are no longer held. You can clear the barrier breakpoint in the action point options dialog box by clicking on **Clear** at the bottom of the action points dialog box.

Defining Evaluation Points

You can define *evaluation points*, points in your program where TotalView evaluates a code fragment. The fragment can include special commands to stop a process and its relatives. Thus, you can use evaluation points to set *conditional breakpoints* of varying complexity. You can also use evaluation points to test potential fixes for your program.

Note: We recommend that you stop a process before setting an evaluation point. This ensures that the evaluation point is set in a stable context in the program.

You can define an evaluation point at any source line that generates executable code (marked with boxed line number in the tag field). If you display assembler or source interleaved with assembler in the process window, you can also define evaluation points on machine-level instructions.

As part of defining an evaluation point, you provide the code fragment to be evaluated. You can write the code fragment in C, Fortran, or Assembler.

Note: Assembler support is currently available only on the Alpha Digital UNIX and AIX operating systems. Compiled expressions must be enabled to use assembler constructs.

At each evaluation point, the code fragment in the evaluation point is executed *before* the code on that line. Typically, the program then executes the program instruction at which the evaluation point is set. But your code fragment can modify this behavior:

- It can include a branching instruction (such as GOTO in C or Fortran). The instruction can transfer control to a different point in the target program, enabling you to code and test program patches.
- It can contain a built-in statement. These special TotalView statements define breakpoints, process barrier points, and countdown breakpoints within the code fragment. By including them within other statements that you code, you can define conditional breakpoints. For more information on these statements, refer to Table 17, “Built-In Statements That Can Be Used in Expressions,” on page 219.

TotalView evaluates code fragments in the context of the target program. This means that you can refer to program variables and pass control to points in the target program.

Note: For complete information on what you can include in code fragments, refer to “Writing Code Fragments” on page 218.

Evaluation points modify only the processes that you are debugging. They do not permanently modify the source program or create a permanent patch in the executable. If you save the evaluation points for a program, however, TotalView reapplies them whenever you start a debugging session for that program. To save your evaluation points, refer to “Saving Action Points in a File” on page 215.

Setting Evaluation Points

To set an evaluation point:

1. Dive into the tag field for an instruction in the process window. TotalView displays the action point options dialog box.
2. Select the **EVAL** (**Evaluate Expression**) button.
3. Select the button (if it’s not already selected) for the language in which you will code the fragment.
4. Select the evaluation text box and enter the code fragment to be evaluated. Use the field editor commands as required. For information on supported C, Fortran, and Assembler language constructs, refer to “Writing Code Fragments” on page 218.
5. For multiprocess programs, decide whether to share the evaluation point among all processes in the program’s share group. By default, the **Share Action Point in All Related Processes** is selected for multiprocess programs, but you can override this by deselecting the checkbox.
6. Select the **OK** button to confirm your changes. If the code fragment has an error, TotalView displays an error message. Otherwise, TotalView processes the code, closes the dialog box, and places an EVAL icon in the tag field.

Setting Conditional Breakpoints

To set a conditional breakpoint, complete steps 1 to 4 of "Setting Evaluation Points" on page 206. Here are some examples of conditional breakpoints and the code fragments that you would need to supply in step 4:

- To define a breakpoint that is reached whenever a variable **i** is greater than 20 but less than 25:

  ```
  if (i > 20 && i < 25)
        $stop;
  ```

- To define a breakpoint that is reached every 10th time the **$count** statement is executed:

  ```
  $count 10
  ```

- To define a breakpoint with a more complex expression, consider this one:

  ```
  $count i * 2
  ```

  When the variable **i** equals 4, the process stops the 8th time it executes the **$count** statement. After the process stops, the expression is reevaluated. If **i** now equals 5, the next stop occurs after the process executes the **$count** statement 10 more times.

Then, complete steps 5 and 6 of "Setting Evaluation Points" on page 206.

For complete descriptions of the **$stop** and **$count** statements, refer to "Built-In Statements" on page 219.

Patching Programs

You can use expressions in evaluation points to patch your code. Specifically, you can use the **goto** (C) and **GOTO** (Fortran) statements to jump to another point in your program's execution.

You can patch programs in two ways:

- You can patch out pieces of code so they are not executed by the program.
- You can patch in new pieces of code to be executed by the program.

In many cases, you correct an error in a program, so you need to use both types of patching. You patch out the incorrect lines of code and patch in the corrections.

Conditionally Patching Out Code

For example, suppose a section of your C program dereferences a null pointer:

```
1  int check_for_error (int *error_ptr)
2  {
3      *error_ptr = global_error;
4      global_error = 0;
5      return (global_error != 0);
6  }
```

In this example, the caller of the **check_for_error** function assumes that passing 0 as the value of **error_ptr** is allowed. The code should allow null values of **error_ptr**, but line 3 dereferences a null pointer.

To correct this error, you can patch in code that checks for a null pointer. To do so, you set an evaluation point on line 3 and specify the following code fragment in the evaluation point:

```
if (error_ptr == 0) goto 4;
```

If the value of **error_ptr** is null, line 3 is not executed.

Patching In a Function Call

As an alternative, you can patch in a **printf** statement that displays the value of **global_error**. To do so, you create an evaluation point on line 4 and specify the following code fragment:

```
printf ("global_error is %d\n", global_error);
```

In this case, the code fragment is executed before the code on line 4, that is, before **global_error** is set to 0.

Correcting Code

In this final example, there is a coding error—the maximum value is returned instead of the minimum value:

```
1  int minimum (int a, int b)
2  {
3      int result;   /* Return the minimum */
4      if (a < b)
5        result = b;
6      else
7        result = a;
8      return (result);
9  }
```

To correct this error, you can set an evaluation point on line 4 and specify the following code fragment to correct the program's **if** statement.

```
if (a < b) goto 7; else goto 5;
```

Interpreted vs. Compiled Expressions

On most platforms, TotalView executes interpreted expressions. TotalView can also execute compiled expressions on the Alpha Digital UNIX and AIX platforms. On these platforms, compiled expressions are enabled by default.

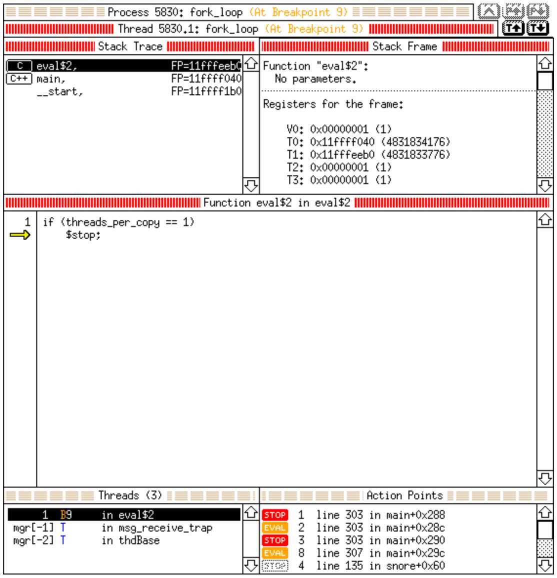
You can enable or disable compiled expressions using Xresources or command-line options. Refer to "totalview*compileExpressions: {true | false}" on page 267. See Appendix B, "Operating Systems," on page 321 to find out how TotalView handles expressions on specific platforms.

Interpreted expressions

- TotalView sets a breakpoint in your code and executes the evaluation point. Since TotalView is executing the expression, interpreted expressions run slower than compiled expressions. With multiprocess programs, interpreted expressions can run more slowly because processes may be waiting serially for the debugger to execute the expression. With remote debugging, interpreted expressions can run more slowly because the debugger, not the debugger server (**tvdsvr**), is executing the expression.
- If the expression contains **$stop** or **$count**, TotalView terminates the evaluation of the expression and stops the process. Thus, if you use **$stop** or **$count**, they should be at the end of your expression because TotalView stops evaluating the expression at that point.

Compiled expressions

- TotalView compiles, links and patches the expression into the target process. To do this, TotalView replaces an instruction with a branch instruction, relocates the original instruction, and appends the expression. Then the code is executed by the target process, so conditional breakpoints can execute very fast.
- If the expression contains **$stop** or **$count**, TotalView stops the execution of the process in the compiled expression, so you can single step through it and continue executing the expression as you would the rest of your code. See Figure 81.


Process 5830: fork_loop (At Breakpoint 9)
Thread 5830.1: fork_loop (At Breakpoint 9)
Stack Trace
Stack Frame
Function eval$2 in eval$2
Threads (3)
Action Points

```
C   eval$2,          FP=11fffeeb0
C++ main,            FP=11ffff040
    __start,         FP=11ffff1b0
```

```
Function "eval$2":
  No parameters.
Registers for the frame:

    V0: 0x00000001 (1)
    T0: 0x11ffff040 (4831834176)
    T1: 0x11fffeeb0 (4831833776)
    T2: 0x00000001 (1)
    T3: 0x00000001 (1)
```

```
if (threads_per_copy == 1)
    $stop;
```

```
      1  B9      in eval$2
mgr[-1]  T       in msg_receive_trap
mgr[-2]  T       in thdBase
```

```
STOP  1  line 303 in main+0x288
EVAL  2  line 303 in main+0x28c
STOP  3  line 303 in main+0x290
EVAL  8  line 307 in main+0x29c
STOP  4  line 135 in snore+0x60
```

Figure 81. Stopped Execution of Compiled Expression

Defining and Using Event Points

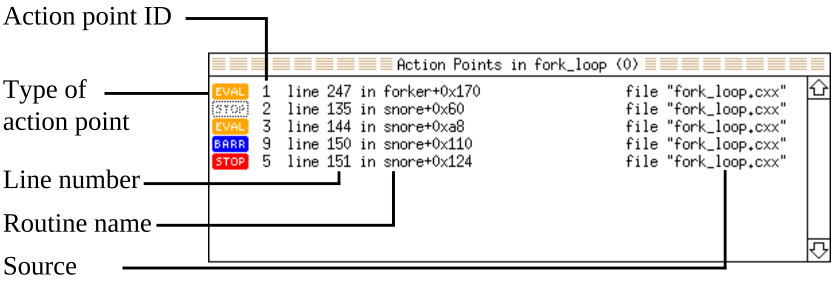
TotalView does not currently support placing event points in your program.

Controlling Action Points

TotalView provides three methods of controlling action points: the action points window, the action points pane in the process window and the action point options dialog box.

Displaying the Action Points Window

The action points window displays a summary of the action points that are set in your program. To display this window, display the **STOP/BARR/EVAL/GIST** submenu and select the **Open Action Points Window (b)** command. The action points window appears, as shown in Figure 82.



```
Action Points in fork_loop (0)
EVAL  1  line 247 in forker+0x170        file "fork_loop.cxx"
STOP  2  line 135 in snore+0x60          file "fork_loop.cxx"
EVAL  3  line 144 in snore+0xa8          file "fork_loop.cxx"
BARR  9  line 150 in snore+0x110         file "fork_loop.cxx"
STOP  5  line 151 in snore+0x124         file "fork_loop.cxx"
```

Figure 82. Action Points Window

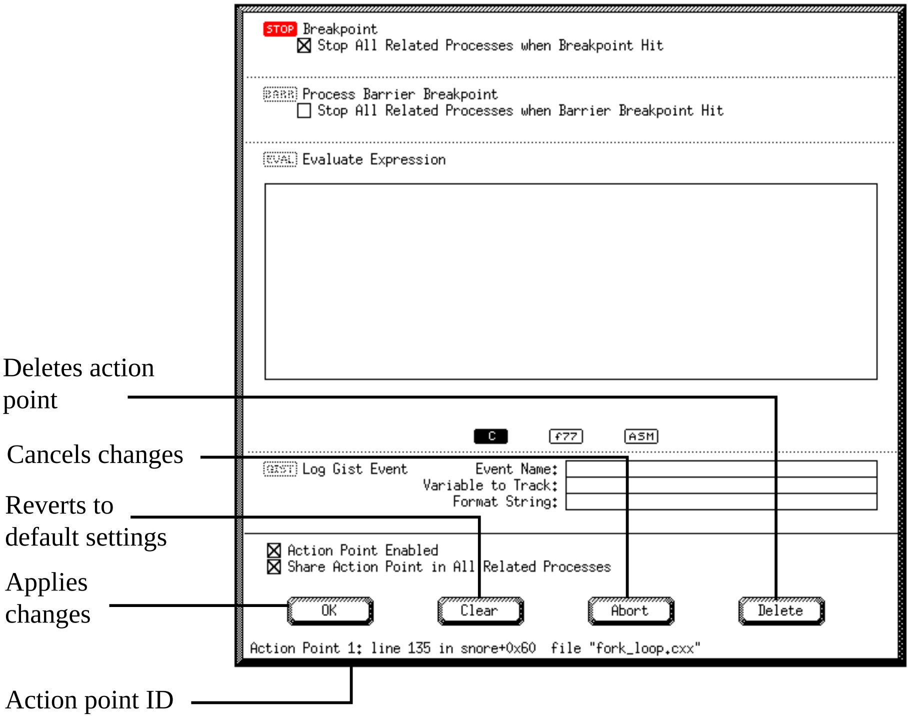
Note: The list of action points displayed in the action points window is the same as shown in the action points pane in the process window.

If you dive into an action point in the action point list, TotalView displays the line of source code containing the action point in the source code pane of the process window.

Tip: Action points make it easier to navigate your source files. You can define disabled breakpoints in your code and dive into the breakpoint to quickly display the corresponding source code in the process window. Thus, breakpoints can act like bookmarks in your program.

Displaying the Action Point Options Dialog

The action point options dialog box lets you set and control an action point in your program. To display this dialog box, dive into the tag field beside a source line or an instruction. TotalView displays the dialog box, illustrated in Figure 83.



STOP Breakpoint
☒ Stop All Related Processes when Breakpoint Hit

BARR Process Barrier Breakpoint
□ Stop All Related Processes when Barrier Breakpoint Hit

EVAL Evaluate Expression

GIST Log Gist Event — Event Name: / Variable to Track: / Format String:

☒ Action Point Enabled
☒ Share Action Point in All Related Processes

Action Point 1: line 135 in snore+0x60 file "fork_loop.cxx"

Deletes action point

Cancels changes

Reverts to default settings

Applies changes

Action point ID

Figure 83. Action Point Options Dialog Box

Commands for Controlling Action Points

You can take the following actions to control the use of action points in your program:

Delete — Permanently removes the action point.

Disable — Keeps the definition for the action point but ignores it during execution.

Enable — Makes the action point active during execution.

Suppress — Keeps the definition for the action point, ignores it during execution, and prevents creation of additional action points.

Unsuppress — Makes the action point active during execution and allows creation of additional action points.

Table 15 shows how to control action points with the process window, action point options dialog, and the action points window.

Table 15. Clearing, Disabling, Enabling, Suppressing, and Unsuppressing Action Points

Action	**Breakpoints and Process Barrier Breakpoints**	**Evaluation Point**	**Event Point**
Deleting	Select the STOP or BARR sign in the tag field. Or Select the **Delete** button in the action point options dialog.	Select the **Delete** button in the action point options dialog.	**Note:** Event points are not currently supported.
	To clear all breakpoints, process barrier points, and evaluation points, go to the process window or action points window, display the **STOP/BARR/EVAL/GIST** submenu, and select the **Clear All STOP, BARR, & EVAL** command.		
Disabling[1]	Deselect **Action Point Enabled** in the action point options dialog. Or Select the STOP or BARR sign in the action points window.	Select the EVAL sign in the tag field. Or Deselect **Action Point Enabled** in the action point options dialog. Or Select the EVAL sign in the action points window.	
Enabling	Select the dimmed **STOP**, **BARR** or **EVAL** sign in the process or action points window. Or Select **Action Point Enabled** in the action point options dialog.		

Table 15. Clearing, Disabling, Enabling, Suppressing, and Unsuppressing Action Points (Continued)

Action	**Breakpoints and Process Barrier Breakpoints**	**Evaluation Point**	**Event Point**
Suppressing[2]	To suppress all action points, display the **STOP/BARR/EVAL/GIST** submenu and select the **Suppress All Action Points (^D)** command.		
Unsuppressing	To unsuppress all action points, display the **STOP/BARR/EVAL/GIST** submenu and select the **Unsuppress All Action Points (^E)** command.		

[1] Disabling an action point does not clear it. TotalView remembers that an action point exists for the line, but ignores it as long as it is disabled. For evaluation points, TotalView keeps the definition in case you want to use it again later.

[2] When you suppress action points, you disable them. In addition, you cannot update any existing action points or create new ones.

Saving Action Points in a File

You can save the action points for each program you debug in a file. By doing so, you will not have to set action points each time you start a new TotalView session. When you save action points, TotalView names the file *program***.TVD.breakpoints**, where *program* is the name of your program.

Tip: To save action points, display the **STOP/BARR/EVAL/GIST** submenu and select the **Save All Action Points** command from the process window. The debugger places the action points file in the same directory as your program.
If you know that you always want to save your action points before you exit from TotalView, you can set an X Window System resource to do this. Refer to “totalview*autoSaveBreakpoints: {true | false}” on page 265. Alternatively, you can use the **–sb** option each time you start the debugger, as described in “TotalView Command Syntax” on page 287.

Once you create an action points file, TotalView automatically loads the file each time you invoke the debugger. TotalView uses the same search paths as it does to locate source files. If you prefer to suppress this behavior, you can set an X resource (see “totalview*autoLoadBreakpoints: {true | false}” on page 265) or use the **–nlb** option each time you start the debugger (see “TotalView Command Syntax” on page 287).

Evaluating Expressions

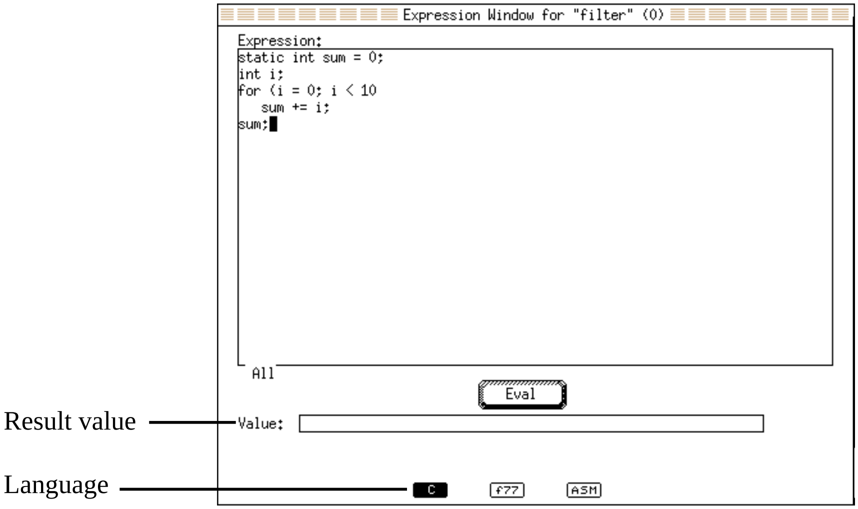
In the TotalView debugger, you can open a window for evaluating expressions in the context of a particular process and evaluate expressions in C, Fortran, or Assembler.

Note: Not all platforms support the use of Assembler constructs; see Appendix C, “Architectures,” on page 333 for details.

To evaluate an expression:

1. Make sure that a process is created, running, or stopped in the process window.
2. Select the **Open Expression Window (e)** command from the process window. An expression evaluation window appears.
3. Select the button (if it is not already selected) for the language in which you will write the code.
4. Select the Expression box and enter the code fragments to be evaluated using the field editor commands. For a description of the supported language constructs, see “Writing Code Fragments” on page 218.

   The last statement in the code fragment can be a free-standing expression; you don’t have to assign the expression’s return value to a variable. Figure 84 shows a sample expression.



```
static int sum = 0;
int i;
for (i = 0; i < 10
   sum += i;
sum;
```

Result value

Language

Figure 84. Sample Expression Window

5. Select the **Eval** button. If TotalView finds an error, it positions the cursor on the incorrect line and displays an error message. Otherwise, it interprets (or on some platforms, compiles and executes) the code, and displays the value of the last expression in the Expression box in the Value field.

While the code is being executed, you can't modify anything in the window because it is suspended. If execution takes a long time, notice that TotalView displays diagonal lines across the window, indicating that the window is temporarily inaccessible.

Since code fragments are evaluated in the context of the target process, the stack variables are evaluated according to the currently selected stack frame. If the fragment reaches a breakpoint (or stops for any other reason), the expression window remains suspended. Assignment statements can affect the target process because they can change the value of a variable in the target process.

You can use the expression window in many different ways, but here are two examples:

- Expressions can contain loops, so you could use a **for** loop to search an array of structures for the entry containing a particular field set to a certain value. In this case, you use the loop index at which the value is found as the last expression in the expression evaluation window.
- You can call subroutines from the expression window, so you could test and debug a single routine in your program without building a test program to call the routine.

Once you have selected and edited an expression in the window, you cannot use a keyboard equivalent (**q**) to exit from the window because the field editor is still active. To exit, display the menu and select the **Close Window** command or press Shift-Return.

Writing Code Fragments

You can use code fragments in evaluation points and in the expression evaluation window. This section describes the intrinsic variables, built-in statements and language constructs supported by TotalView.

Intrinsic Variables

The TotalView expression system supports built-in variables that allow you to access special thread and process values. All of the variables are of type 32-bit integer, which is type **<int>** or **<long>** on most platforms. The variables are not lvalues, so you cannot assign to them or take their addresses. Table 16 lists the intrinsic variable names and their meanings.

Table 16. Intrinsic Variables

Name	Meaning
$tid	Returns the TotalView-assigned thread ID. When referenced from a process, generates an error.
$systid	Returns the system-assigned thread ID. When referenced from a process, generates an error.
$pid	Returns the process ID.
$nid	Returns the node ID.
$clid	Returns the cluster ID.
$duid	Returns the TotalView-assigned Debugger Unique ID (DUID).
$processduid	Returns the DUID of the process.

Note: **$nid**, **$clid**, **$duid**, and **$processduid** are implemented for interpreted expressions only.

Intrinsic variables allow you to create thread specific breakpoints from the expression system. For example, using the **$tid** intrinsic variable and the **$stop** built-in operation, you can create a thread specific breakpoint as follows:

```
if ($tid == 3)
   $stop;
```

This would cause TotalView to stop the process only if thread 3 evaluated the expression. You can also create complex expressions using intrinsic variables:

```
if ($pid != 34 && $tid > 7)
   printf (“Hello from %d.%d\n”, $pid, $tid);
```

Built-In Statements

TotalView provides a set of built-in statements that you can use when writing code fragments. The statements are available in all languages, and are shown in Table 17.

Table 17. Built-In Statements That Can Be Used in Expressions

Statement	Use
$stopthread	Sets a thread-level breakpoint. The thread that executes this statement stops, but all other threads in the process continue to execute. If the target system does not support asynchronous stop, this executes as a **$stopprocess**.
$stop **$stopprocess**	Sets a process-level breakpoint. The process that executes this statement stops, but other processes in the program group continue to execute.
$stopall	Sets a program-group-level breakpoint. *All* processes in the program group stop when any thread or process in the group executes this statement.
$countthread *expression*	Sets a thread-level countdown breakpoint. When any thread in a process executes this statement for the number of times specified by *expression*[1], it stops. The other threads in the process continue to execute. If the target system does not support asynchronous stop, this executes as a **$countprocess**.
$count *expression* **$countprocess** *expression*	Sets a process-level countdown breakpoint. When any thread in a process executes this statement for the number of times specified by *expression*, the process stops. The other processes in the program group continue to execute.

Table 17. Built-In Statements That Can Be Used in Expressions (Continued)

Statement	Use
$countall *expression*	Sets a program-group-level countdown breakpoint. *All* processes in the program group stop when any process in the group executes this statement for the number of times specified by *expression*.
$hold **$holdprocess**	Holds the current process. If all other processes in the group are already held in breakpoint state at this **eval** point, then all will be released. If other processes in the group are running, they *continue to run.*
$holdstopall **$holdprocessstopall**	Exactly like **$hold**, except any processes in the group which are running are *stopped.* Note that the other processes in the group are *not* automatically held by this call -- they are just stopped.
$holdthread	Freezes the current thread leaving other threads running. See (later sections) for more information on threads.
$holdthreadstop **$holdthreadstopprocess**	Exactly like **$holdthread** except it *stops* the *process*. The other processes in the group are left running.
$holdthreadstopall	Exactly like **$holdthreadstop** except it stops the entire group.
$visualize(*expression*[**,***slice*]**)**	Visualizes the data specified by *expression* and modified by the optional *slice*. *Expression* and *slice* must be written in the syntax of the code fragment's language. The expression can be any valid expression that yields a data-set (after modification by slice) that can be visualized. The slice is a quoted string containing a slice expression. For more information on how to use $visualize in an expression, see "Visualizing Data in Expressions" on page 239.

[1] A thread evaluates *expression* when it executes the **$count** statement for the first time, and it must evaluate to a positive integer. A thread reevaluates **$count** only when it results in a breakpoint. Then, the process' internal counter for the breakpoint is reset to the value of *expression*. The internal counter is stored in the process and shared by all threads in that process.

C Constructs Supported

When writing code fragments in C, keep these guidelines in mind.

Syntax

- C-style (/* comment */) and C++-style (// comment) comments are permitted. For example:

  ```
  // This code fragment creates a temporary patch
  i = i + 2;        /* Add two to i */
  ```

- Semicolons can be omitted when no ambiguity would result.
- Dollar signs ($) in identifiers are permitted.

Data Types and Declarations

- Data types permitted: **char**, **short**, **int**, **float**, **double**, and pointers to any primitive type or any named type in the target program.
- Only simple declarations are permitted. The **struct**, **union**, and array declarations are *not* permitted.
- References to variables of any type in the target program are permitted.
- Unmodified variable declarations are considered local. References to them override references to similarly named global variables and other variables in the target program.
- (Compiled evaluation points only) The **global** declaration makes a variable available to other evaluation points and expression windows in the target process.
- (Compiled evaluation points only) The **extern** declaration references a global variable that was or will be defined elsewhere. If the global variable has not yet been defined, TotalView displays a warning.
- **Static** variables are local and persist even after an evaluation point has been evaluated.
- For static and global variables, expressions that initialize data as part of the variable declaration are performed only the first time the code fragment is evaluated. Local variables are initialized each time the code fragment is evaluated.

Statements

- Permitted statements: assignment, **break**, **continue**, **if**/**else** structures, **for**, **goto**, and **while**.

- With the **goto** statement, you can define and branch to symbolic labels. These labels are considered local to the window. As an extension, you can also refer to a line number in the target program. This line number refers to the *tag field* number of the source code line. Here's a **goto** statement that causes the program to branch to source line number 432 of the target program:

  ```
  goto 432;
  ```

- Function calls are permitted, but structures cannot be passed to a function.
- Type casting is permitted.
- All operators are permitted, with these limitations:
  - The conditional operator **?:** is not supported.
  - The **sizeof** operator can be used for variables, but not data types.
  - The **(***type***)** operator cannot cast to fixed-dimension arrays using C cast syntax.

Fortran Constructs Supported

When writing code fragments in Fortran, keep these guidelines in mind.

Syntax

- Syntax is free-form. No column rules apply.
- One statement is allowed per line, and one line is allowed per statement.
- The space character is significant and sometimes required. (Some Fortran 77 compilers ignore *all* space characters, wherever they are coded.) For example:

  ```
  Valid            Invalid

  DO 100 I=1,10    DO100I=1,10
  CALL RINGBELL    CALL RING BELL
  X .EQ. 1         X.EQ.1
  ```

  GOTO, GO TO, ENDIF, and END IF are all allowed. But ELSEIF is not; use ELSE IF.

- Comment lines can be defined in several formats. For example:

  ```
  C I=I+1
  ```

```
/*
I=I+1
J=J+1
ARRAY1(I,J)= I * J
*/
```

Data Types and Declarations

- Data types permitted: INTEGER (assumed to be long), REAL, DOUBLE PRECISION, and COMPLEX.
- Implied data types are *not* permitted.
- Only simple declarations are permitted. The COMMON, BLOCK DATA, EQUIVALENCE, STRUCTURE, RECORD, UNION, and array declarations are *not* permitted.
- References to variables of any type in the target program are permitted.

Statements

- Permitted statements: assignment, CALL (to subroutines, functions, and all intrinsic functions except CHARACTER functions in the target program), CONTINUE, DO, GOTO, IF (including block IF, ENDIF, ELSE, and ELSE IF), and RETURN (but not alternate RETURN).
- As an extension to the GOTO statement, you can refer to a line number in the target program. This line number refers to the *tag field* number of the source code line. For example, this GOTO statement causes the program to branch to source line number 432 of the target program:

  ```
  GOTO $432;
  ```

  The dollar sign is *required* before the line number to distinguish the tag field number from a statement label.

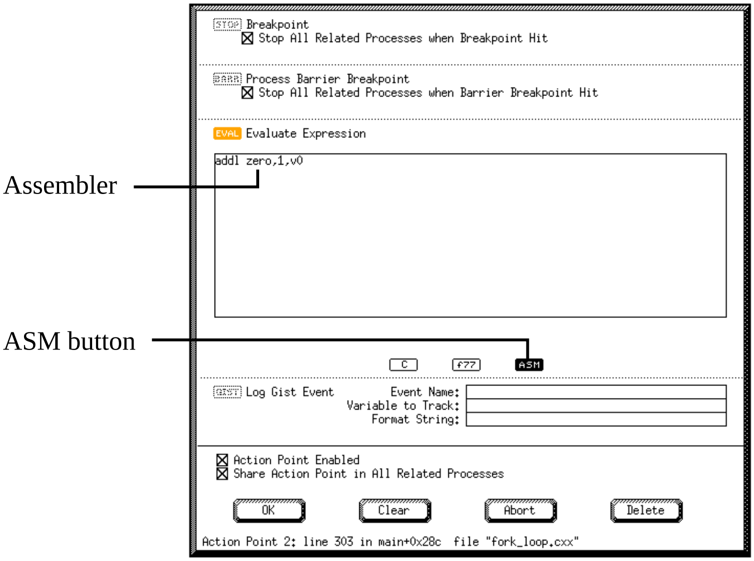
- All expression operators are supported except CHARACTER operators and the logical operators .EQV., .NEQV., and .XOR..
- Subroutine function and entry definitions are not permitted.
- Fortran 90 array syntax is not supported.
- Fortran 90 pointer assignment (the => operator) is not supported.
- Calling Fortran 90 functions which require assumed shape array arguments is not supported.

Writing Assembler Code

On Alpha Digital UNIX and RS/6000 IBM AIX operating systems, TotalView supports the use of assembler code in EVAL points, conditional breakpoints, and the expression window.

Note: If you want to use assembler constructs, you must enable compiled expressions. See “Interpreted vs. Compiled Expressions” on page 209 for instructions.

To indicate that an expression in the breakpoint or expression windows is an assembler expression, click on the **ASM** button in the expression window as shown in Figure 85.



Breakpoint
☒ Stop All Related Processes when Breakpoint Hit

Process Barrier Breakpoint
☒ Stop All Related Processes when Barrier Breakpoint Hit

Evaluate Expression

```
addl zero,1,v0
```

Log Gist Event
Event Name:
Variable to Track:
Format String:

☒ Action Point Enabled
☒ Share Action Point in All Related Processes

Action Point 2: line 303 in main+0x28c file "fork_loop.cxx"

Figure 85. **ASM** Button in Expression Window

Assembler expressions are written in the TotalView Assembler Language. In the TotalView Assembler Language, instructions are written identically to the native assembler language, but the operators available to construct expressions in instruction operands and the set of available pseudo-operators is common across all architectures.

The TotalView Assembler accepts instructions using the same mnemonics recognized by the native assembler and recognizes the same names for registers that native assemblers recognize. Some architectures provide extended mnemonics that do not correspond exactly with machine instructions. Normally, these extended mnemonics represent important, special cases of instructions, or provide for assembling short, commonly used sequences of instructions. The TotalView Assembler recognizes such extended mnemonics as long as they meet both of the following criterion:

- They assemble to exactly one instruction, and
- The relationship between the operands of the extended mnemonics and the fields in the assembled instruction code is a simple one-to-one correspondence

In TotalView Assembler Language, labels are indicated as *name***:**, appearing at the beginning of a line. Labels may appear alone on a line. Symbols available for use include any labels defined in the assembler expression and all program symbols.

The set of operators available for use in the TotalView Assembler are indicated in Table 18.

Table 18. TotalView Assembler Operators

Operators	**Definition**
hi16 (*expr***)**	Low 16 bits of operand *expr*
lo16 (*expr***)**	High 16 bits of operand *expr*
hi32 (*expr***)**	High 32 bits of operand *expr*
lo32 (*expr***)**	Low 32 bits of operand *expr*
“*text***”**	Text string, 1-4 characters long, is right justified in a 32-bit word
+	Plus
–	Minus (also unary)
*****	Times
/	Quotient

Table 18. TotalView Assembler Operators (Continued)

Operators	**Definition**
#	Remainder
&	Bitwise and
^	Bitwise xor
!	Bitwise or not (also unary - bitwise not)
|	Bitwise or
<<	Left shift
>>	Right shift
(_expr_**)**	Grouping

The set of pseudo-operations available for use in the TotalView Assembler are listed in Table 19:

Table 19. TotalView Assembler Pseudo Ops

Pseudo Ops	**Definition**
$debug [**0** | **1**]	Internal debugging option. With no operand, toggle debugging; **0** => turn debugging off; **1** => turn debugging on
$ptree	Internal debugging option. Print assembler tree
$stop **$stopprocess**	Stop the process
$stopall	Stop the program group
$stopthread	Stop the thread

Table 19. TotalView Assembler Pseudo Ops (Continued)

Pseudo Ops	Definition
$hold **$holdprocess**	Hold the process
$holdstopall **$holdprocessstopall**	Hold the process and stop the program group
$holdthread	Hold the thread
$holdthreadstop **$holdthreadstopprocess**	Hold the thread and stop process
$holdthreadstopall	Hold the thread and stop the program group
$long_branch *expr*	Branch to location *expr*, using a single instruction in an architecture independent way, without requiring the use of any registers
align *expr* [**,** *expr*]	Align location counter to an operand 1 alignment; use operand 2 (or zero) as the fill value for skipped bytes
byte *expr* [**,** *expr*] …	Place *expr* values into a series of bytes
half *expr* [**,** *expr*] …	Place *expr* values into a series of 16 bit words
word *expr* [**,** *expr*] …	Place *expr* values into a series of 32 bit words
quad *expr* [**,** *expr*] …	Place *expr* values into a series of 64 bit words
float *expr* [**,** *expr*] …	Place *expr* values into a series of floats
double *expr* [**,** *expr*] …	Place *expr* values into a series of doubles
string *string*	Place *string* into storage

Table 19. TotalView Assembler Pseudo Ops (Continued)

Pseudo Ops	**Definition**
ascii *string*	Same as *string*
asciz string	Zero terminated string
zero *expr*	Fill *expr* bytes with zeros
fill *expr* **,** *expr* **,** *expr*	Fill storage with operand 1 objects of size operand 2, filled with value operand 3
org *expr* [**,** *expr*]	Set location counter to operand 1 use operand 2 (or zero) to fill skipped bytes
def *name*,*expr*	Define a symbol with *expr* as it's value
name=*expr*	Same as **def** *name*,*expr*
lsym *name*,*expr*	Same as **def** *name*,*expr* but allows redefinition of a previously defined name

bss *name*,*expr*[,*expr*]	Define *name* to represent operand 2 bytes of storage in the **bss** section with alignment operand; default alignment depends on the size:
	if size >= 8 then 8 else
	if size >= 4 then 4 else
	if size >= 2 then 2 else 1

comm *name*,*expr*	Define name to represent *expr* bytes of storage in the **bss** section; *name* is declared global; alignment is as in **bss** without an alignment argument
lcomm *name*,*expr*[,*expr*]	Identical to **bss**
global *name*	Declare *name* as global
text	Assemble code into text section (code)
data	Assemble code into data section (data)

Table 19. TotalView Assembler Pseudo Ops (Continued)

Pseudo Ops	**Definition**
equiv *name*,*name*	Make operand 1 be an abbreviation for operand 2

CHAPTER 9: Visualizing Data

The TotalView Visualizer is part of a suite of software development tools for debugging, analyzing, and tuning the performance of programs. It works with the TotalView debugger to create graphic images of array data in your programs. This lets you see your data graphically as you debug your programs.

The visualizer is implemented as a self-contained process. It can be launched directly by TotalView to visualize data as you debug your programs. Alternatively, you can run the visualizer from the command line to visualize data dumped to a file in a previous TotalView session.

You interact with TotalView to choose *what* you want to visualize and *when* the snapshot of your data should be grabbed. You interact with the visualizer to choose *how* you would like your data to be displayed.

Note: The TotalView Visualizer is not available on all platforms.

This chapter explains how to use the Visualizer with TotalView to visualize array data. In this chapter, you will learn:

- How the visualizer works
- Launching the Visualizer from TotalView
- Types of data that TotalView can visualize
- Visualizing data from the TotalView variable window
- Visualizing data in expressions at breakpoints and in the expression window
- What the Visualizer's windows do
- Changing settings from the directory window

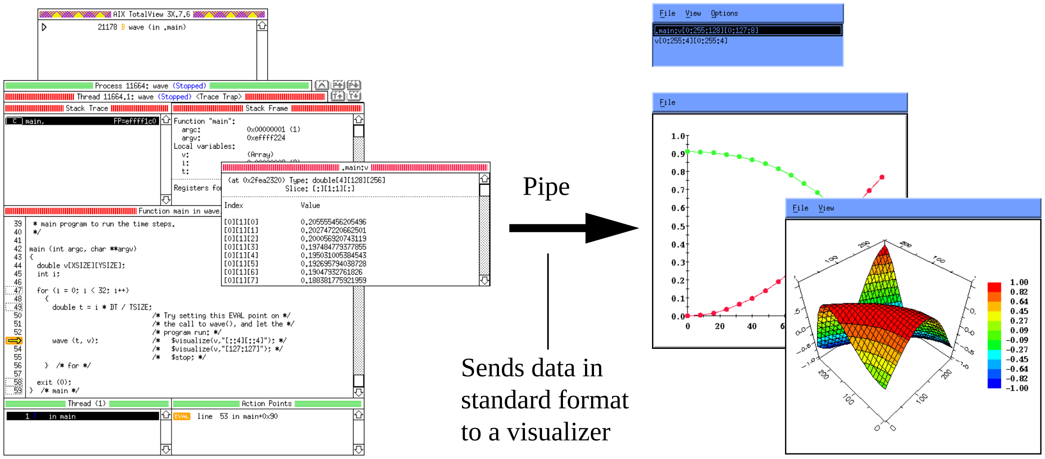
- Data-set formatting for the visualizer
- Methods of visualization
- Changing displays of data
- Manipulating displays of data
- Launching the Visualizer from the command line
- Launching the Visualizer from a third party debugger
- Adapting third party visualizers to TotalView

How the Visualizer Works

There are two sides to using the TotalView Visualizer; extracting data from the program being debugged, and displaying the data graphically. The TotalView debugger handles the first of these, extracting the data and marshaling it into a standard format that it sends down a pipe. The Visualizer then reads the data from this pipe and displays it for your analysis. Figure 86 shows how this is implemented.

TotalView – Extracts data from an array

The TotalView Visualizer – Displays the array data graphically



```
Function "main":
  argc:          0x00000001 (1)
  argv:          0xeffff224
Local variables:
  v:             (Array)
  i:
  t:
```

```
(at 0x2fea2320) Type: double[4][128][256]
                Slice: [:][1:1][:]

Index            Value

[0][1][0]        0.205555456205496
[0][1][1]        0.202747220662501
[0][1][2]        0.200056920743119
[0][1][3]        0.197484779377855
[0][1][4]        0.195031005384543
[0][1][5]        0.192695794038728
[0][1][6]        0.19047932761826
[0][1][7]        0.188381775921959
```

```
 * main program to run the time steps.
 */

main (int argc, char **argv)
{
  double v[XSIZE][YSIZE];
  int i;

  for (i = 0; i < 32; i++)
    {
      double t = i * DT / TSIZE;
                                  /* Try setting this EVAL point on */
                                  /* the call to wave(), and let the */
                                  /* program run: */
      wave (t, v);                /*   $visualize(v,"[::4][::4]"); */
                                  /*   $visualize(v,"[127:127]"); */
                                  /*   $stop; */
    }  /* for */

  exit (0);
}  /* main */
```

```
.main:v[0:255:128][0:127:8]
v[0:255:4][0:255:4]
```

Pipe

Sends data in standard format to a visualizer

Figure 86. TotalView Visualizer Connection

This split between the TotalView Debugger and the Visualizer also allows for different implementations of the visualizer. It means that you interact with TotalView to choose *what* you want to visualize and *when* you want to grab a snapshot of your data. You interact with the Visualizer to choose *how* to display your data.

You can send data directly from the TotalView Debugger to the TotalView Visualizer while you are debugging your program. You can send data from TotalView directly to a third party visualizer that allows you to write your own visualizer program, or adapt an interface for visualization with a third-party product. Finally, you can launch the TotalView Visualizer from the command line using data you have already saved to a file. Figure 87 shows these relationships.

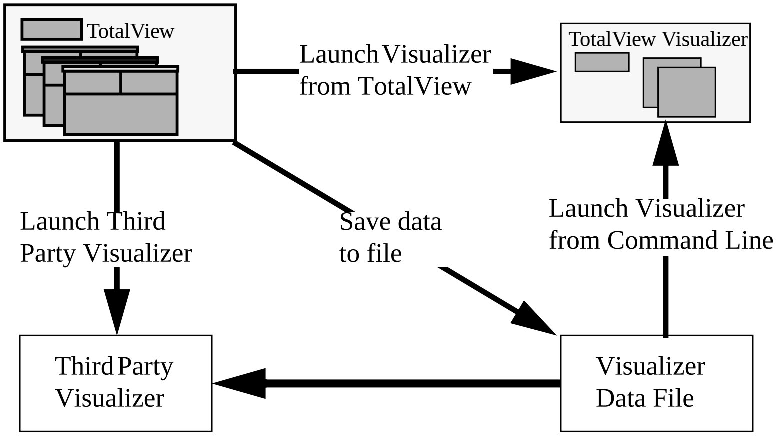

TotalView
Launch Visualizer from TotalView
TotalView Visualizer
Launch Third Party Visualizer
Save data to file
Launch Visualizer from Command Line
Third Party Visualizer
Visualizer Data File

Figure 87. TotalView Visualizer Relationships

Configuring TotalView to Launch the Visualizer

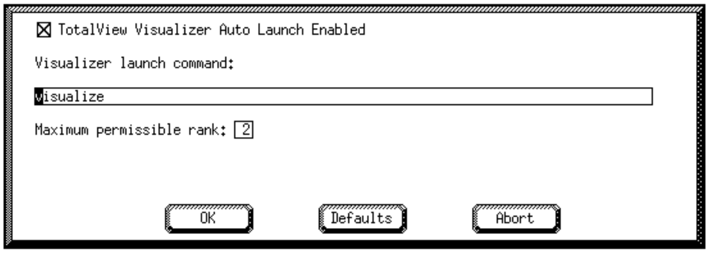
When TotalView launches the Visualizer, it pipes data to standard input of the Visualizer so you can visualize data-sets as your program creates them.

TotalView automatically launches the visualizer when it is requested in a variable, breakpoint, or expression window. You can configure TotalView to set the following:

- Whether or not visualization is enabled.
- The shell command used by TotalView to launch the Visualizer.
- The maximum number of dimensions of an array that TotalView will export to the Visualizer.

If you disable visualization, all attempts to use the visualizer are silently ignored. This can be useful if you want to execute some code containing evaluation points that do visualization, but do not want to disable all the evaluation points individually.

To change the Visualizer launch options interactively, select the **Visualizer Launch Window** from the root window. A dialog box appears, as shown in Figure 88.



☒ TotalView Visualizer Auto Launch Enabled

Visualizer launch command:

visualize

Maximum permissible rank: 2

Figure 88. The Visualizer Launch Window

To enter your choices, do the following:

1. Change the auto launch option. The TotalView Visualizer is set to enable visualization and launch the visualizer automatically by default. If you do not want it to launch automatically and disable visualization, clear the **TotalView Visualizer Auto Launch Enabled** checkbox.
2. If you want the visualizer to use a customized command when it starts, enter it in the **Visualizer launch command** box.
3. Change the maximum permissible rank. Edit the value (the supported range is **1** through **16**) if you plan to save the data that you export from TotalView to a file or display it in a different visualizer.

   The maximum permissible rank (the default is **2**), ensures that data exported by TotalView is suitable for display in the TotalView Visualizer which displays only two dimensions of data. This limit does not apply to data saved in files, or to visualizers that can display more than two dimensions of data.
4. Clicking on the **Defaults** button sets the options to the defaults. Note that this reverts to the standard defaults even if you have used an X resource to change the settings on start-up.
5. If you want to use these settings, press **Return** or click on the **OK** button. To abandon your edits, click on the **Abort** button.

If you disable visualization or change the visualizer launch string while a visualizer is running, TotalView closes the pipe to the visualizer. If you enable visualization again, a new visualizer process will be launched the next time you visualize something.

You can change the shell command that TotalView uses to launch the visualizer by editing the Visualizer launch string. This is useful if you want to run a different visualizer, or if you want to save visualization data to a file for viewing later. For example, you can save the file with the following visualizer launch string:

```
cat > your_file
```

Later, you can visualize the file with one of the following (equivalent) commands:

```
% visualize –persist < your_file

% visualize –file your_file
```

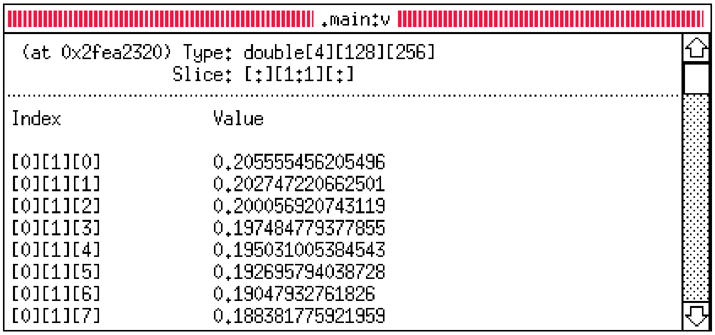
You can set the visualizer launch options automatically when TotalView starts, by setting X resources. For details, see Chapter 11, “X Resources,” on page 263.

Data Types that TotalView Can Visualize

The data you select for visualization as a single entity is called a *data-set*. Each data-set passed from TotalView to the Visualizer is tagged with a numeric identifier to tell the Visualizer whether this is a new data-set, or an update to an existing data-set. TotalView creates the identifier from the program, base address and type of the data selected for visualization. This ensures that when you visualize the “same” data by different methods, the same set of images is updated. Note that this causes the visualization of a stack variable at different recursion levels or call paths, to appear as separate images instead of updates to an existing image.

By default, TotalView restricts the type of data it can visualize to one and two dimensional arrays of character, integer, or floating point data. These must be located in memory, not in registers.

You can visualize arrays with more dimensions by using an array slice expression to extract an array with fewer dimensions. Figure 89 shows how a three dimensional variable has been sliced to two dimensions by selecting a single index in the middle dimension to permit visualization.

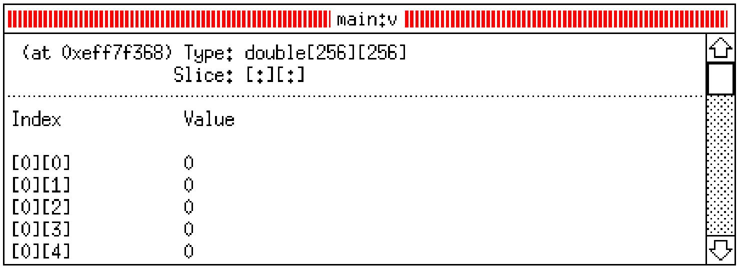


```
.main:v

 (at 0x2fea2320) Type: double[4][128][256]
                Slice: [:][1:1][:]

Index                Value

[0][1][0]            0.205555456205496
[0][1][1]            0.20274722066250l
[0][1][2]            0.200056920743119
[0][1][3]            0.197484779377855
[0][1][4]            0.195031005384543
[0][1][5]            0.192695794038728
[0][1][6]            0.19047932761826
[0][1][7]            0.188381775921959
```

Figure 89. A Three Dimensional Array Sliced to Two Dimensions

Visualizing Data from the Variable Window

The simplest way to visualize data from TotalView is by using the variable window. For details on the variable window, see Chapter 7, “Examining and Changing Data,” on page 147. Open a variable window on the array of interest and stop program execution where the array values are those you would like to visualize. At this point, the TotalView variable window should show the current array values in a text format as shown in the example in Figure 90.



```
 (at 0xeff7f368) Type: double[256][256]
                 Slice: [:][:]

Index              Value

[0][0]             0
[0][1]             0
[0][2]             0
[0][3]             0
[0][4]             0
```

Figure 90. Variable Window

You can edit the type and slice expressions in the variable window to select the precise data you want to visualize. You can display subsections of arrays, which are called *slices*, to limit the volume of data you examine at one time. See “Displaying Array Slices” on page 172. Limiting the volume of data increases the speed of the visualizer.

With the desired array (or array slice) displayed in the variable window, select the **Visualize (v)** command from the window menu to launch the visualizer program and send it the array data you want to visualize. The first visualize command launches the visualizer program (if needed) and creates the initial data window display. Subsequent **Visualize** commands send updated data values and cause the visualizer to update its display.

You can visualize laminated data pane displays, using the **visualize** command. See “Visualizing a Laminated Data Pane” on page 179. The process or thread index forms one of the dimensions of the visualized data. By default therefore, you are restricted to visualizing scalar or vector information. If you do not want the process or thread index as one of the dimensions of your visualization, you can use a non-laminated display instead and visualize it.

Visualizer data displayed through a variable window is not automatically updated as you step through your program. You must explicitly request an update by reissuing the **Visualize (v)** command in variable window.

Visualizing Data in Expressions

You can use TotalView's expression system to visualize data with the **$visualize** built-in function. You can use it to:

- Visualize several different variables from a single expression
- Visualize variables in the expression evaluation window
- Visualize one or more variables from an evaluation point

The **$visualize** built-in function takes two parameters separated by a comma.

$visualize (*array* [**,** *slice_string*]**)**

The first parameter *array* is an expression that specifies a data-set for visualization. The second parameter *slice_string* is optional. If present, it is a quoted string containing a constant slice expression that modifies the data-set specified by the first parameter. The following examples assume that your program contains a two dimensional array called my_array.

Table 20. **$visualize** examples for C and Fortran

C	**Fortran**
$visualize (my_array);	$visualize (my_array)
$visualize (my_array,"[::2][10:15]")	$visualize (my_array,'(11:16,::2)')
$visualize (my_array,"[12][:]");	$visualize (my_array,'(:,13)')

The first example simply visualizes the whole array. The second example selects every second element in the major dimension of the array, and also clips the minor dimension to all elements in the given (inclusive) range. The third example reduces the visualized data-set to a single dimension, by selecting a single sub-array.

You may have to use a cast expression to let TotalView know the dimensions of the variable you want to visualize. The following shows a procedure that passes a two dimensional array parameter, without specifying the extent of the major dimension.

```
void my_procedure (double my_array[][32])
{ /* procedure body */ }
```

Attempts to visualize **my_array** directly will fail because the first dimension is not specified. The following cast expression defines the dimensions of the array, and allows you to visualize it.

```
$visualize (*(double[32][32]*)my_array);
```

You can use the **$visualize** built-in statement in an expression in the expression window or by adding an expression to a breakpoint to create an evaluation point. But note that any evaluation point or expression in the expression window that includes an instance of **$visualize** cannot be compiled. Instead, the TotalView debugger interprets these statements. See "Defining Evaluation Points" on page 205 for information about compiled and interpreted expressions.

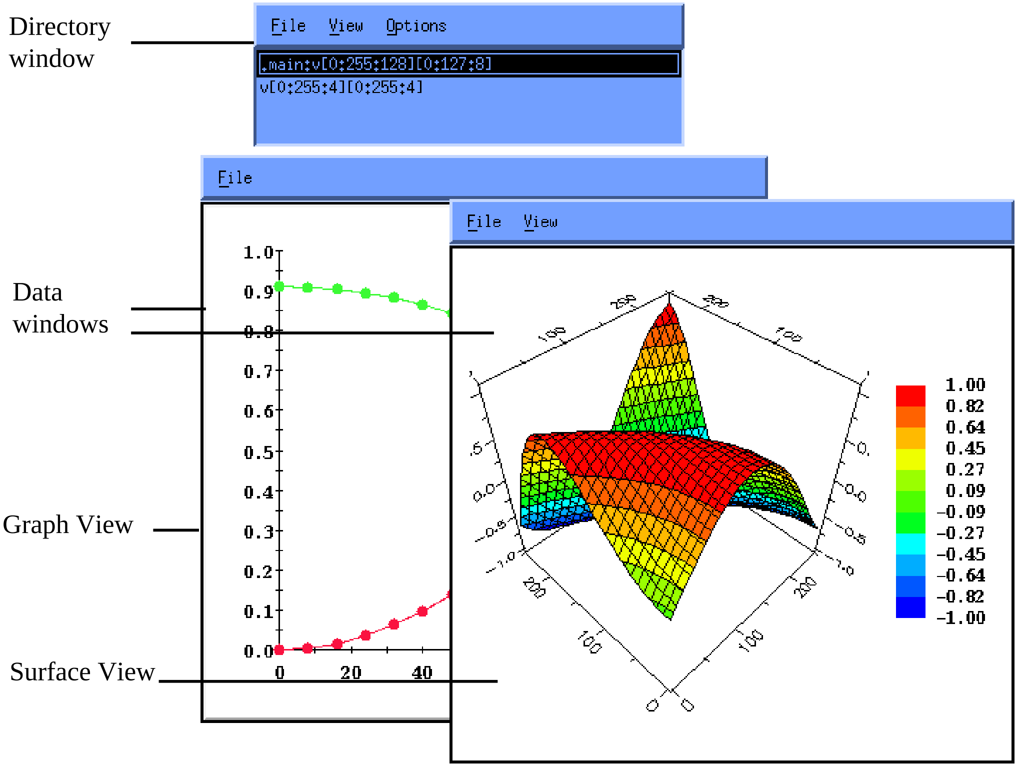
Using **$visualize** in the expression window is a handy technique to refine the array and slice arguments or to update the Visualizer display of several arrays simultaneously.

Visualizer Animation

Using the **$visualize** built-in statement in an evaluation point expression is a powerful technique to provide an animated display of your data. When used in an evaluation point, the **$visualize** statement forces the Visualize program to update its display of the array argument every time the evaluation point is reached by program execution. By setting an evaluation point using **$visualize** at program statements which change the values of array elements, you can create a visual animation of the array as the program executes.

The TotalView Visualizer

The Visualizer is implemented as a self-contained process. You can launch it directly from TotalView while you are debugging your programs. Or, you can launch it from the command line to visualize data you saved to a file. The Visualizer can read data-sets on its standard input stream, or from a file. The Visualizer windows are shown in Figure 91.



```
.main:v[0:255:128][0:127:8]
v[0:255:4][0:255:4]
```

Figure 91. Visualizer Windows

The Visualizer is a Motif application, so you may change some of the Visualizer settings using X resources. See “Visualizer X Resources” on page 283.

The Visualizer has two types of windows:

- A directory window

  A single main window lists the data-sets that you can visualize. You can interact with the directory window to set global options and to create views of your data-sets.

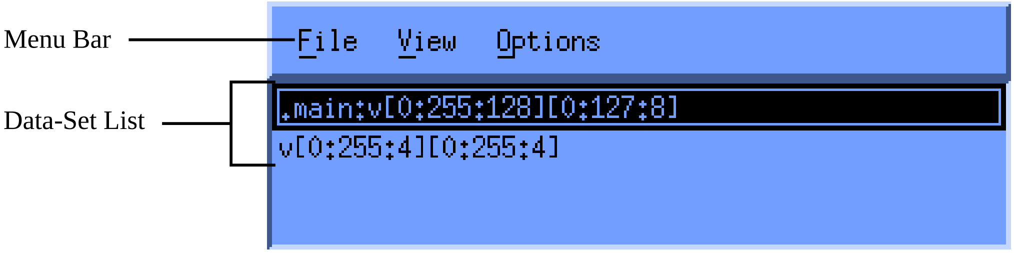
- Data windows

  The *data windows* contain images of the data-sets. By interacting with a data window, you can change its appearance and set options on viewing its data-set. Using the directory window, you can open several data windows on a single data-set to get different views of the same data.

Directory Window

The directory window contains a list of the data-sets you can display in the Visualizer. You can create these data-sets during your debugging session or from a file See Figure 92.

Whenever TotalView passes the Visualizer a new data-set the Visualizer updates the list of data-sets in the directory window.



```
.main:v[0:255:128][0:127:8]
v[0:255:4][0:255:4]
```

Figure 92. Sample Visualizer Directory Window

You can select a data-set by left-clicking on it. You can select only one data-set at a time. Right-clicking in the data-set list displays the **View** menu. From this menu, you can select **Graph** or **Surface** visualization.To delete a data-sets from the list, click on it then display the **File** menu and select **Delete**. Updates to existing data-sets do not alter the list.

You can automatically visualize the selected data-set by left-clicking in the data-set then pressing Return. You can also double-left-click in the data-set list to select and auto-visualize a data-set.

For a list of the menu and command choices from the directory window, see Table 21.

Table 21. Directory Window Menu Commands

Menu	Command	Meaning
File	**Delete**	Deletes the currently selected data-set. It removes the data-set from the data-set list and destroys any data windows displaying it.
	Exit	Closes all windows and exits the Visualizer.
View	**Graph**	Creates a new graph window. See "Graph Data Window" on page 247 for more detail.
	Surface	Creates a new surface window. See "Surface Data Window" on page 249 for more detail.
Options	**Auto Visualize**	This item is a toggle. When enabled, the Visualizer automatically visualizes new data-sets as they are read. See section 4 for information on how the visualization method is selected.

Data Windows

Data windows display graphical images of your data. An example of two different types of data window is show in Figure 93.

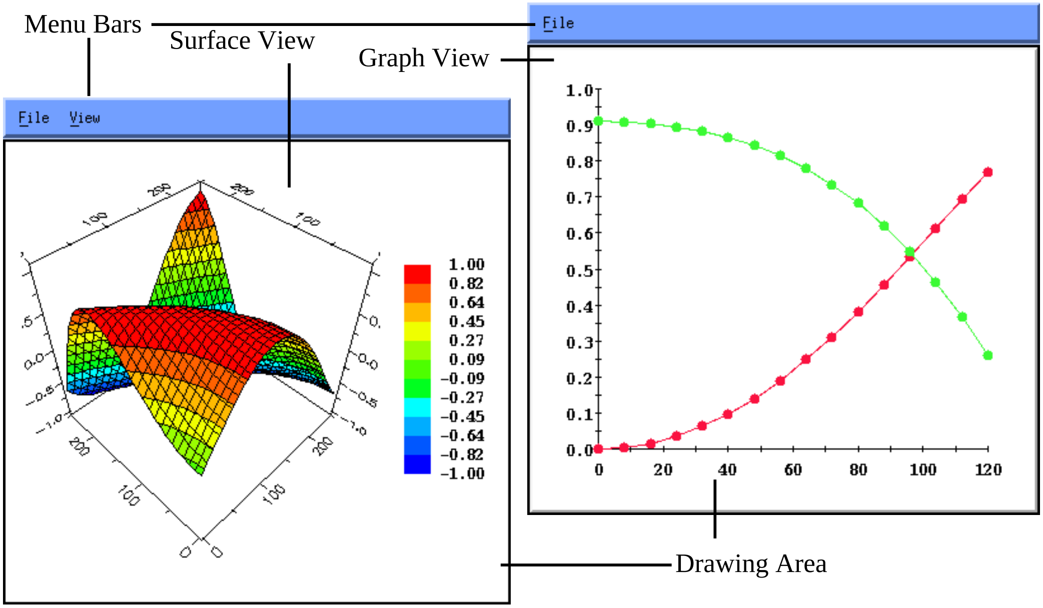


Figure 93. Sample Visualizer Data Windows

All data windows contain a menu bar and a drawing area. The data window title, as displayed by the window manager, is its data-set identification.

The **File** menu on the menu bar is the same for all data windows. Any other items on the menu bar are specific to particular types of data window. The common data window menu commands are described in Table 22.

Table 22. Data Window Menu Commands

Menu	Command	Meaning
File	**Directory**	Raises the directory window to the front of the desktop. If the directory window is currently minimized, it is restored.
	New Base Window	Creates a new data window using the same visualization method and data-set as the current data window. This helps you to create several views of a data-set using the same visualization method.
	Options	Pops-up a window of viewing options. This window consists of a *control* area and an *action* area. The control area is specific to the type of data window. The action area contains three buttons as follows: • **OK** — Apply any changes and pop down the options window. • **Apply** — Apply the options settings in the control area, but leave the options window up. • **Cancel** — Pop down the options and discard any changes in the control area. You can also cancel any changes you have made in the control area by closing the options window.
	Delete	Deletes the data window's data-set from the data-set list. This also destroys any other data windows viewing the data-set.
	Close	Closes the data window.

The drawing area displays the image of your data. You can interact with the drawing area to alter the view of your data. For example, in the surface view, you can rotate the graph to view it from different angles. You can also get the value and indices of the data-set element nearest the cursor by left-clicking on it. A pop-up a window displays the information. For details on this and other ways to manipulate the surface view, see Table 27, “Surface Data Window Manipulations,” on page 253.

Views of Data

Different types of data-sets require different graphical views to display their data. For example, a graph is more suitable for displaying one dimensional data-sets or two dimensional data-sets where one of the dimensions has a small extent. But a surface view is necessary for displaying a two dimensional data-set.

You can manually choose a visualization method for a given data-set or you can let the Visualizer choose one for you. The Visualizer chooses a method based on the following criteria:

1. If any data windows are currently displaying the data-set, they are raised to the top of the desktop. If any of these windows is minimized, they are restored.
2. If no data windows exist for the data-set, but the data-set has been visualized previously, the Visualizer creates a new data window using the most recent visualization method.
3. If the data-set has never been visualized, the Visualizer chooses a method of display based on the type of the data-set. Methods that can’t visualize the data-set are ruled out. The Visualizer then chooses one of the remaining methods, based on an internal scoring system that measures how well a given data-set matches an ideal data-set for each method.

The Visualizer can automatically choose a visualization method and create a new data window when it reads a new data-set. When the data-set is an update to an existing data-set, the Visualizer uses the method last used to visualize the data. You can enable and disable this feature from the **Options** menu in the TotalView Visualizer directory window.

Graph Data Window

The graph window displays a two dimensional graph of one or two dimensional data-sets. If the data-set is two dimensional, multiple graphs are displayed. When you first create a graph window on a two dimensional data-set, the Visualizer uses the dimension with the larger number of elements for the X axis. It then draws a separate graph for each sub-array in the dimension with the smaller number of elements. If this choice is not correct, you can transpose the data. Graph visualization does not favor two dimensional data-sets with large extents in both dimensions as this gives a very cluttered display.

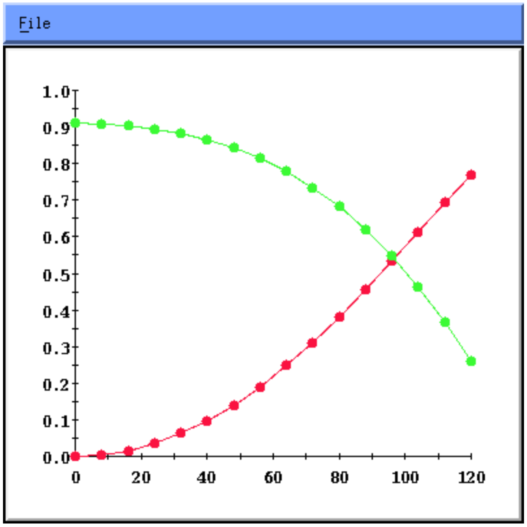


Figure 94. Visualizer Graph Data Window

You can display graphs with markers for each element of the data-set, with lines connecting data-set elements, or with both lines and markers as shown in Figure 94. See “Displaying Graphs” on page 248 for more details. Multiple graphs are displayed in different colors. The X axis of the graph is annotated with the indices of the long dimension. The Y axis shows you the data value.

You can scale and translate the graph, or pop up a window displaying the indices and values for individual data-set elements. See “Manipulating Graphs” on page 248 for details.

Displaying Graphs

The graph options dialog lets you control how to display the graph. You can bring up this dialog box by displaying the **File** menu and selecting the **Options** command. See Table 23 for details.

Table 23. Graph Data Window Options Dialog

Toggle	Meaning
Lines	Toggles the display of lines connecting data-set elements.
Points	Toggles the display of markers for each data-set element.
Transpose	Toggles the choice of dimension to map onto the X axis of the graph for two dimensional data-sets.

Manipulating Graphs

You can get a detailed view of part of the graph by using the keyboard and mouse with the focus in the drawing area. You can control scaling and translation separately, or both together with a zoom. You can also query individual element values. See Table 24 for details.

Table 24. Graph Data Window Manipulations

Action	Description
Scale	Press the Control key and hold down the middle mouse button. Move the mouse down to zoom in on the center of the drawing area, or up to zoom out.
Translate	Press the Shift key and hold down the middle mouse button. Moving the mouse drags the graph.
Zoom	Press the Control key and hold down the left mouse button. Drag the mouse button to create a rectangle that encloses the area of interest. The area is then translated and scaled to fit the drawing area.
Reset View	Press **r** to reset all translation and scaling. This resets the view of the graph to the initial state.

Table 24. Graph Data Window Manipulations (Continued)

Action	Description
Query	Hold down the left mouse button near a graph marker. A window pops up displaying the data-set element’s indices and value.

Figure 95 shows a graph view of two dimensional random data. It is created by selecting **Points** and deselecting **Lines** in the graph data window options dialog box.

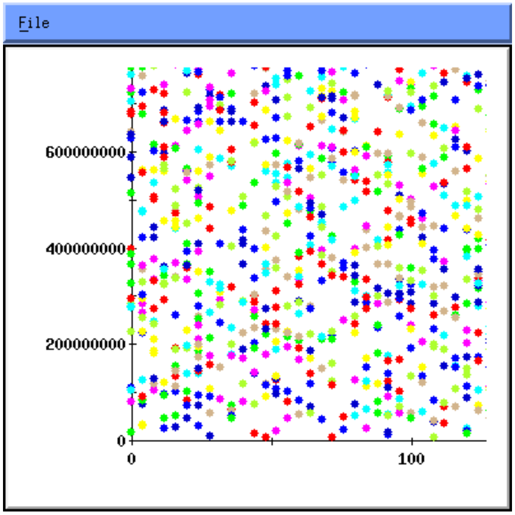


Figure 95. Display of Random Data

Surface Data Window

The surface data window displays two dimensional data-sets as a surface in two or three dimensions. The data-set’s array indices map to the first two dimensions (X and Y axes) of the display. Figure 96 shows a two dimensional map, where the data-set values are shown using only the **Zone** option to demarcate ranges of element values. For a zone map with contour lines, turn the **Zone** and **Contour** settings on and **Mesh** and **Shade** off.

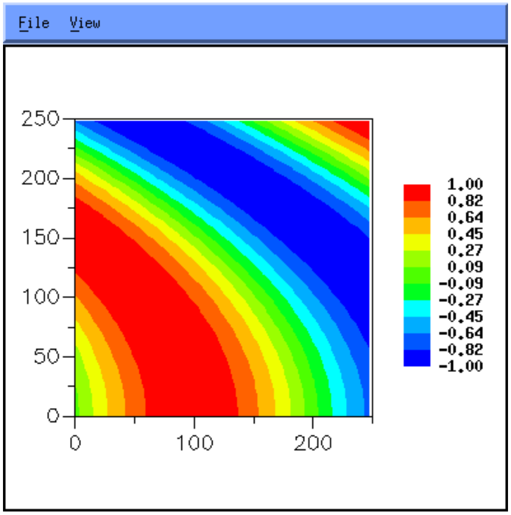


Figure 96. Two Dimensional Surface Visualizer Data Display

You can display random data by selecting only the **Zone** setting and turning **Mesh**, **Shade**, and **Contour** off. The display shows where the data is located and allows you to click on it to get the values of the various points.

Figure 97 shows a three dimensional surface which maps element values to the height (Z axis).

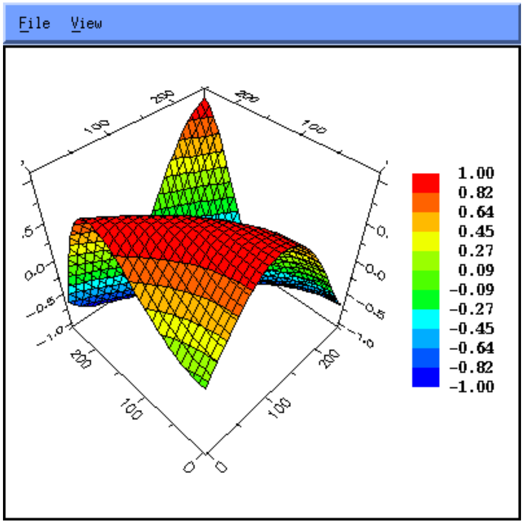


Figure 97. Three Dimensional Surface Visualizer Data Display

Displaying Surface Data

You have several options to help you control the display of the surface data. They are available in the options dialog box. In the data window, display the **File** menu and select the **Options** command. A dialog box appears with choices shown in Table 25.

Table 25. Surface Data Window Options

Toggle	Meaning
Mesh	Toggles the *mesh* option. When this option is set, the surface is displayed in three dimensions, with the X-Y grid projected onto the surface. When neither this option nor the *shade* option are set, the surface is displayed in two dimensions (See Figure 96).
Shade	Toggles the *shade* option. When this option is set, the surface is displayed in three dimensions and shaded either in a “flat” color to differentiate the top and bottom sides of the surface, or in colors corresponding to the value if the *zone* option is also set. When neither this option nor the *mesh* option are set, the surface is displayed in two dimensions (See Figure 96).
Contour	Toggles the *contour* option. When this option is set, contour lines are displayed demarcating ranges of element values.
Zone	Toggles the *zone* option. When this option is set, the surface is displayed in colors demarcating ranges of element values.
Auto Reduce	Toggles the *auto-reduce* option. When this option is set, the surface displayed is derived by averaging over neighboring elements in the original data-set. This speeds up the visualization method by reducing the resolution of the surface. Clear this option if you want to accurately visualize all data-set elements.

The **Auto Reduce** option allows you to choose between viewing all your points of data, which takes longer to appear in the display, or viewing an averaging of data over a number of nearby points, which appears in the display much faster. The default for **Auto Reduce** is on so your display appears faster.

You can reset the viewing parameters to those in effect when the Visualizer first came up. The **View** menu in the data window lets you reset the viewing parameters. Choose **Reset View** (**r**) from the **View** menu in the data window.

Table 26. Surface Data Window Menu Commands

Menu	Command	Meaning
View	**Reset View**	Restores all translation, rotation and scaling. This resets the view of the surface to the initial state and enlarges the display slightly.

Manipulating Surface Data

You can rotate a three dimensional surface to change the viewing angle, so you can see parts of the surface that are hidden from some viewing angles, or get a detailed view of part of the surface. When you click and hold the middle mouse button in the drawing area, then drag the mouse. The image changes to a wireframe bounding box of the surface which moves with the mouse. You can rotate the view in two dimensions simultaneously, or select a single axis at a time to rotate. When you let go of the button, you can see the graph from the new, selected vantage point.

In addition to rotating the graph, you can manipulate it several other ways, as shown in Table 27. You can display the indices and values of individual data-set elements in a pop up window. You can control scaling and translating separately, or together with a zoom. You can query the values of individual elements. And you can reset the view to what it was when you started.

Table 27. Surface Data Window Manipulations

Action	Description
Rotate	Hold down the middle mouse button and drag the mouse to freely rotate the surface. You can also press the **x**, **y**, or **z** keys to select a single axis of rotation.
Scale	Press the Control key and hold down the middle mouse button. Move the mouse down to zoom in on the center of the drawing area, or up to zoom out.

Table 27. Surface Data Window Manipulations (Continued)

Action	Description
Translate	Press the Shift key and hold down the middle mouse button. Moving the mouse drags the surface.
Zoom	Press the Control key and hold down the left mouse button. Drag the mouse button to create a rectangle that encloses the area of interest. The area is then translated and scaled to fit the drawing area.
Reset View	Press **r** to reset translation and scaling. This does not reset the rotation.
Query	Hold down the left mouse button near the surface. A window pops up displaying the nearest data-set element’s indices and value.

Launching the Visualizer from Command Line

To start the Visualizer from the shell command line, use the following syntax:

```
% visualize [options]
```

where options include:

–file *filename*	Reads data from *filename* instead of reading from standard input.
–persist	Continues to run after encountering an EOF on standard input. Without this option, the Visualizer exits as soon as it reads all of the data from standard input.

By default, the Visualizer reads its input data sets from its standard input stream and exits when it reads an EOF on standard input. When started by TotalView, the Visualizer normally reads its data from a pipe, ensuring that the Visualizer exits when TotalView does. If you want the Visualizer to continue to run after it exhausts all input from the standard input stream, you should invoke the Visualizer with the **–persist** option.

If you want to read data from a file, invoke the Visualizer with the **–file** *filename* option. For example:

```
% visualize –file my_data_set_file
```

When you visualize data-sets from a file, the Visualizer reads all the data-sets in the file. This means that the images you see are of the last versions of the data-sets in the file.

Visualize supports the generic X toolkit command line options. For example, you can start the Visualizer with the directory window minimized by using the **–iconic** option. Your system manual page for the X server or the *The X Window System User's Guide*, by O'Reilly & Associates lists the generic X command line options in detail.

You can also customize the Visualizer by setting X resources in your resource files or on the command line with the **–xrm** *resource_setting* option. The available resources are described in Chapter 12, "TotalView Command Syntax," on page 287. Use of X resources to modify the default behavior of TotalView or the TotalView Visualizer is described in greater detail in Chapter 11, "X Resources," on page 263.

Adapting a Third Party Visualizer to the TotalView Debugger

TotalView passes a stream of data-sets to the Visualizer encoded in the format described below. You can write your own Visualizer or adapt an interface to a third-party Visualizer by parsing this format. However, before doing this, you should be aware of some assumptions in the design of the interface:

- The data-set encoding assumes that TotalView and the Visualizer are running on the same machine architectures, meaning that word lengths, byte order and floating-point representations are identical. Note that there is sufficient information in the data-set header to detect when this is not the case (with the exception of floating-point representation), but no simple way of describing any required translations.
- TotalView transmits data-sets down the pipe in a simple unidirectional flow. There is no handshaking protocol in the interface. This requires the Visualizer to be an eager reader on the pipe. If the Visualizer does not read eagerly, the pipe will back up and block TotalView.

The format of a data-set is included in the TotalView distribution in a header file named **include/visualize.h** in the TotalView installation directory. Each data-set is encoded with a fixed-length header followed by a stream of array elements. The header contains the following fields.

Table 28. Data-Set Header Fields

Field	Meaning
vh_magic	Contains **VIS_MAGIC**, a symbolic constant to provide a check that this *is* a data-set header and that byte order is compatible.
vh_version	Contains **VIS_VERSION**, a symbolic constant to provide a check that the reader understands the protocol.
vh_id	Contains the data-set id. Every data-set in a stream of data-sets is numbered with a unique id so that updates to a previous data-set can be distinguished from new data-sets.

Table 28. Data-Set Header Fields (Continued)

Field	Meaning
vh_title	Contains a plain text string of length **VIS_MAXSTRING** that annotates the data-set.
vh_axis_order	Contains one of the constants **vis_ao_row_major** or **vis_ao_column_major**.
vh_type	Contains one of the constants **vis_signed_int**, **vis_unsigned_int**, or **vis_float**.[1]
vh_item_length	Contains the length (in bytes) of single element of the array.
vh_item_count	Contains the total number of elements to be expected.
vh_effective_rank	Contains the number of dimensions that have an extent larger than **1**.
vh_dims	Contains information on each dimension of the data-set. This includes a base, count and stride. Only the count is required to correctly parse the data-set. The base and stride only give information on the valid indices in the original data.[2]

[1] Types in the data-set are encoded by a combination of the **vh_type** field and the **vh_item_length** field. This allows the format to handle arbitrary sizes of both signed and unsigned integers, and floating point numbers.

The **vis_float** constant corresponds to the default floating point format (usually, IEEE) of the target machine. The Visualizer does not handle values other than the default on machines that support more than one floating point format.

Although a three byte integer is expressible in the Visualizer's data-set format, it is unlikely that the Visualizer will handle one. The Visualizer only handles data types that correspond to the C data types permitted on the machine where the Visualizer is running.

Similarly, the long double type varies significantly depending on the C compiler and target machine. Therefore, visualization of the long double type is unlikely to work if you run the Visualizer on a machine that is different from the one where you extracted the data.

In addition, you need to be aware of these data type differences if you write your own visualizer and plan to run it on a machine that is different from the one where you extract the data.

2. Note that all **VIS_MAXDIMS** of dimension information is included in the header, even if the data has fewer dimensions.

The data following the header is a stream of consecutive data values of the type indicated in the header. Consecutive data values in the input stream correspond to adjacent elements in **vh_dims[0]**.

You can verify that your reader’s idea of the size of this type is consistent with TotalView by checking that the value of the n_bytes field of the header matches the product of the size of the type and the total number of array elements.

CHAPTER 10: Troubleshooting

This chapter describes how to solve common problems that you might encounter while using TotalView. Refer to Table 29.

Table 29. Symptoms and Solutions

Symptom	**Possible Solutions**
Windows don't appear or operate correctly	• Your DISPLAY environment variable is not set correctly. • The resource "totalview*useTransientFor: {on | off}" on page 281 is not set correctly. Change it from **on** to **off**, or from **off** to **on**. • Start Totalview with the **–grab** command-line option. • Use the **xhost +** command to allow all hosts to access your display.
Pressing Control-C in an **xterm** window causes TotalView to exit	• Start TotalView with the **–ignore_control_c** or **–icc** command-line option.
Source code doesn't appear in source code pane	• Set the search path for directories with the **Set Search Directory (d)** command in the process window.
License manager does not operate correctly	• Set the **LM_LICENSE_FILE** environment variable to the pathname of the TotalView license file. See the *TotalView Installation and Administration Guide* for details.

Table 29. Symptoms and Solutions (Continued)

Symptom	Possible Solutions
Fatal error: Checkout … failed	• Check the value of the **LM_LICENSE_FILE** environment variable. Make sure the value ends with the string **license.dat**. • Make sure the TotalView license manager **lmgrd** is running on the license manager host machine. The name of this machine is listed in the **SERVER** line of your **license.dat** file. • Make sure that the **lmgrd** that is running matches the one which came with your TotalView distribution.
Out of memory error	• Increase the swap space on your machine. For details, see “Swap Space” on page 324. • Increase the data size limit in the C shell. Use the C shell’s **limit** command, such as: `% limit datasize unlimited`
Error creating new process	• Increase the swap space on your machine. For details, see “Swap Space” on page 324. • Increase the number of process slots in your system. See your operating system documentation for details. • Check the **xterm** window to see if the **execve()** call failed, and if it did, set the PATH environment variable. • Make sure that the **/proc** filesystem is mounted on your system. For details, see “Mounting the /proc File System” on page 323.
Error launching process or Attempt to delete the target of an unbound process	• Run your program at the UNIX command line prompt to see if it will load and start executing. When it passes this test, you can run TotalView on your program to debug it. • If the operating system can’t load your program and start it, make sure your program is built for the machine you are debugging on.

Table 29. Symptoms and Solutions (Continued)

Symptom	Possible Solutions
When debugging HPF programs, HPF source code does not appear in the process window; only f77 code appears.	• When compiling HPF program be sure to set both the **–g** and **–Mtotalview** flags when both compiling and linking your programs.

Program behaves differently under TotalView's control	• Make sure your program does not setuid or exec another program which does, for example, **rsh**. Normally, the operating system will not allow a debugger to debug a setuid executable nor allow a setuid system call while a program is being debugged. Often these operations fail silently. To debug setuid programs, login as the target UID before starting TotalView.
	• TotalView uses the SIGSTOP signal to stop processes. On most UNIX systems, system calls can fail with the *errno* set to EINTR when the process receives a SIGSTOP signal. You need to change your code so that it handles the EINTR failure. For example:
	`do {`
	`      n = read(fd,buf,nbytes);`
	`} while (n < 0 && errno == EINTR);`

The TotalView server, **tvdsvr**, fails to start on a remote node.	• Re-edit the server launch command field, click OK, and launch the server again. For information, see "Starting the Debugger Server for Remote Debugging" on page 64.

X resources are not recognized	• Use the **xrdb** command (part of the X Window System) to display the current X resources:
	`xrdb -query`
	• Use the **xrdb** command to load your X resources:
	`xrdb -load $HOME/.Xdefaults`
	• Read the **xrdb** manual page for more information.

Single stepping is slow or TotalView is slow to respond to breakpoints	• Close some of the variable windows that you have open. • The global variables window is open and has a large number of variables. Close the global variables window.

Table 29. Symptoms and Solutions (Continued)

Symptom	Possible Solutions
Other fatal error or Internal error in TotalView	• Report this problem. See “Reporting Problems” on page iv.

CHAPTER 11: X Resources

This chapter provides reference information about the X Window System resources that you can use to customize TotalView or the TotalView Visualizer. You can use these resources in your X resources files (such as **.Xdefaults** on UNIX systems or **decw$sm_general.dat** on VMS systems).

For information on X resources files, refer to the X Window System documentation that came with your machine or the *X Window System User's Guide*, by O'Reilly & Associates (ISBN 1–56592–015–5).

On most UNIX systems, you load your X resources file using the **xrdb** command (part of the X Window System executables). For example:

```
% xrdb –load $HOME/.Xdefaults
```

The default value for each resource in this chapter is shown in **bold**.

TotalView X Resources

You can override some of the resources with command-line options for the **totalview** command, as described in “TotalView Command Syntax” on page 287.

Note: You can specify any of the following X resources on the command line using the “**–***Xresource***=***value*” command line option specified on page 288. For example, to set **totalview*stopAll** to **false**, you could specify the command line option **–stopAll=false**. Note that the string “**totalview***” is omitted from the command line

---

Window Locations

Values for the location of windows are expressed as:

=*width***x***height*+*x*+*y*

where *width* is the width of the window in pixels, *height* is the height of the window in pixels, *x* is the distance from the upper-left corner of the window to the left screen edge in pixels, and *y* is the distance from the upper-left corner of the window to the top screen edge in pixels. A value of **-1** for *x* or *y* indicates that the window should be centered in the screen with respect to the x-axis or y-axis. If desired, you can express *x* or *y* as negative numbers to indicate the distance from the lower-right corner of the window to the bottom screen edge or right screen edge instead of the distance from the upper-left corner. A value of zero (**0**) indicates that TotalView should use the default value. Also, you can supply just the size (*width* and *height*), and TotalView will use the default location (*x* and *y*) with it.

As an example, the expression **=0x0-1+20** uses the default width and height, centers the window horizontally, and places the window 20 pixels down from the top of the screen. The expression **=330x120+20-20** makes the window 330 pixels wide by 120 pixels high and places the window 20 pixels from the left edge of the screen and 20 pixels up from the bottom edge of the screen.

totalview*arrowBackgroundColor: *color*

Sets the background (outline) color of PC arrow to *color*.

Default: **black**

totalview*arrowForegroundColor: *color*

Sets the foreground (inner) color of PC arrow to *color*.

Default: **yellow2**

totalview*autoLoadBreakpoints: {**true** | false}

If true (default), automatically load action points from the file **filename.TVD.breakpoints**. If false, you use the **STOP/BARR/EVAL/GIST** -> **Load All Action Points** command in the process window to load action points.

Override with:	**–lb** option (overrides **false**) **–nlb** option (overrides **true**)

totalview*autoRetraceAddresses: {**on** | off}

If on (default), TotalView will retrace the sequence of dive operations performed in a variable window and recompute a new address for the variable. If off, does not retrace addresses.

totalview*autoSaveBreakpoints: {true | **false**}

If false (default), do not automatically save action points to an action points file when you exit. You use the **STOP/BARR/EVAL/GIST** -> **Save All Action Points** command in the process window to save action points.

Override with:	**–sb** option (overrides **false**) **–nsb** option (overrides **true**)

totalview*backgroundColor: *color*

Sets the general background color to *color*.

Default: **white**

totalview*barrierForegroundColor: *color*

Sets the color of the barrier point icon.

Default: **blue**

totalview*barrierFontForegroundColor: *color*

Sets the color of the font used to show the **H** and **Hold** indicators for held processes.

Default: **blue**

totalview*barrierStopAll: {**true** | false}

Same as **totalview*processBarrierStopAllRelatedProcessesWhenBreakpointHit**.

totalview*blindMouse: {**on** | off}

If on (default), allow "*mouse ahead*," the queuing of mouse clicks (similar to typing ahead in a shell). If off, successive mouse clicks are ignored until TotalView responds to the first mouse click.

totalview*breakFontForegroundColor: *color*

Sets the color of "**B**" state to *color*.

Default: **orange**

totalview*breakpointWindLocation: =*width*x*height*+*x*+*y*

Specifies placement of the first action points window.

Default:

width	*height*	*x*	*y*
columns(70)	**lines(12)**	**335**	**10**

totalview*buttonBackgroundColor: *color*

Sets the button background color to *color*. Defaults to the background color.

totalview*buttonForegroundColor: *color*

Sets the button foreground color to *color*. Defaults to the foreground color.

totalview*chaseMouse: {**on** | off}

If on (default), display dialog boxes at the location of the mouse cursor. If off, display dialog boxes centered in the upper third of the screen.

Override with:	**–chase** option (overrides **off**) **–no_chase** option (overrides **on**)

totalview*compilerVars: {true | **false**}

Alpha Digital UNIX and SGI only. If false (default), TotalView does not show variables created by the Fortran compiler. If true, TotalView shows variables created by the Fortran compiler and the variables in the user's program.

Some Fortran compilers (Digital f90/f77, SGI 7.2 compilers) output debug information which describes variables that the compiler itself has invented for purposes such as passing the length of character*(*) variables. By default TotalView suppresses the display of these compiler generated variables, however you can set **totalview*compilerVars** to true to cause such variables to be displayed. This could be useful if you are looking for a corruption of a run time descriptor or are writing a compiler.

Override with:	**–compiler_vars** option (overrides **false)** **–no_compiler_vars** option (overrides **true**)

totalview*compileExpressions: {**true** | false}

Alpha Digital UNIX and IBM AIX operating systems only. If true (default), TotalView enables compiled expressions. If false, TotalView disables compiled expressions and interprets them instead.

totalview*conditionVariableInfoWindLocation: **=***width***x***height***+***x***+***y*

Specifies placement of the first condition variable information window.

Default:

width	*height*	*x*	*y*
columns(75)	**lines(15)**	**360**	**300**

totalview*cTypeStrings: {true | **false**}

If false (default), use TotalView's type string extensions when displaying the type strings for arrays. If true, use C type string syntax when displaying arrays.

totalview*dataWindLocation: =*width*x*height*+*x*+*y*

Specifies placement of the first variable window.

Default:

width	*height*	*x*	*y*
columns(72)	**max(205, lines(15))**	**-80**	**320**

totalview*displayAssemblerSymbolically: {on | **off**}

If off (default), display Assembler locations as hexadecimal addresses. If on, display Assembler locations as "label+offset."

totalview*DPVMDebugging: {true | **false**}

Digital UNIX only.

If false (default), disables support for debugging the Digital UNIX implementation of Parallel Virtual Machine (DPVM) applications. If true, enables support for debugging DPVM applications.

Override with:	**–dpvm** option (overrides **false**) **–no_dpvm** option (overrides **true**)

totalview*editorLaunchString: *command_string*

Sets the editor launch command string to the specified value. Refer to "Changing the Editor Launch String" on page 122 for more information on the format of *command_string*.

Default: **xterm –e %E +%N %S**

totalview*errorFontForegroundColor: *color*

Sets the color of "**E**", "**Z**", and "**?**" states to *color*.

Default: **red**

totalview*evalForegroundColor: *color*

Sets the color of the EVAL action point signs to *color*.

Default: **orange**

totalview*evalWindLocation: =*width*x*height*+*x*+*y*

Specifies placement of the first expression evaluation window.

Default:

width	*height*	*x*	*y*
columns(83)	**lines(30) + 2**	**-1**	**10**

totalview*eventLogWindLocation: =*width*x*height*+*x*+*y*

Specifies placement of the event log window.

Default:

width	*height*	*x*	*y*
columns(75)	**lines(20)**	**-75**	**-50**

totalview*font: *fontname*

Specifies the font used by the TotalView debugger. Use the X Windows supplied application **xlsfonts** to list the names of available fonts.

Default: **fixed**

totalview*foregroundColor: *color*

Sets the general foreground color (i.e., the text color) to *color*.

Default: **black**

totalview*frameOffsetX: *n*

Sets the horizontal placement offset between windows of the same type, as TotalView places them on the screen. This value is *added* to the default value used by TotalView. If you are using TotalView title bars, use the default.

Default: **0**

totalview*frameOffsetY: *n*

Sets the vertical placement offset between windows of the same type, as TotalView places them on the screen. This value is *added* to the default value used by TotalView. If you are using TotalView title bars, use the default.

Default: **0**

totalview*globalsWindLocation: =*width***x***height***+***x***+***y*

Specifies placement of the global variables window.

Default:

width	*height*	*x*	*y*
columns(62)	**max(205, lines(15))**	**-80**	**10**

totalview*globalTypenames: {**true** | false}

If true (default), specifies that TotalView can assume that type names are globally unique within a program and that all type definitions with the same name are identical. In C++, the standard asserts that this must be true for standard conforming code.

If this option is true, TotalView will attempt to replace an opaque type (**struct foo *p;**) declared in one module, with an identically named defined type (**struct foo { … };**) in a different module.

If TotalView has read the symbols for the module containing the non-opaque type definition, then when displaying variables declared with the opaque type, TotalView will automatically display the variable using the non-opaque type definition.

If false, specifies that TotalView *cannot* assume that type names are globally unique within a program. You should specify this option if your code has multiple different definitions of the same named type, since otherwise TotalView is likely to pick the wrong definition to substitute for an opaque type.

Override with: **–global_types** option (overrides **false**)
–no_global_types option (overrides **true**)

totalview*grabMouse: {on | **off**}

If off (default), do not force keyboard input to dialog boxes. If you're running TotalView with a window manager that is operating in "click-to-type" mode, you should set this resource to "on" or use the **–grab** command-line option.

totalview*helpWindLocation: =*width***x***height*+*x*+*y*

Specifies placement of the help window.

Default:

width	*height*	*x*	*y*
min(screen_width - 10, columns(84))	**min(screen_height - 20, 606)**	**-1**	**-20**

totalview*hpf: {**true** | false}

If true (default, if HPF debugging has been licensed), enables debugging at the HPF source level.

Setting this X resource to **false**, causes TotalView to ignore **.stx** and **.stb** files, and therefore to debug HFP code at the intermediate (Fortran 77) level.

Override with:	**–hpf** option (overrides **false**) **–no_hpf** option (overrides **true**)

totalview*hpfNode: {true **| false}**

If false (default), the node on which an HPF distributed array element resides is not displayed in the process window.

The node display can be toggled in each variable window using the **Toggle Node Display** option in the process window menu.

Override with:	**–hpf_node** option (overrides **false**) **–no_hpf_node** option (overrides **true**)

totalview*inverseVideo: {true **| false}**

If true, enables inverse video display. If false (default), disables inverse video display.

totalview*kccClasses: {**true** | false}

If set to true, (default) TotalView will convert structure definitions output by the KCC compiler into classes that show base classes, and virtual base classes in the same way as other C++ compilers. When set to false, TotalView will not convert structure definitions output by the KCC compiler into classes. Virtual bases will show up as pointers, rather than the data.

Unfortunately, the conversion has to be done by textual matching of the names given to structure members, so can it be confused if you have structure component names that look to TotalView like KCC processed classes. However, the conversion is never performed unless TotalView believes that the code was compiled with KCC, because TotalView has seen one of the tag strings that KCC outputs, or because the user has asked for the KCC name demangler to be used. Also all of the recognized structure component names start with "__", and, according to the C standard, user code should not contain names with this prefix.

Note that under some circumstances it is not possible to convert the original type names because there is no available type definition. For example, it may not be possible to convert "struct __SO_foo" to "struct foo", so in this case the "__SO_foo" type will be shown. This is only a cosmetic problem. (The "__SO__" prefix denotes a type definition for the non-virtual components of a class with virtual bases).

Since KCC outputs no information on the accessibility of base classes ("private", "protected", "public"), TotalView is unable to provide this information.

totalview*mainHSplit: *n*

Same as **totalview*mainHSplit1**.

totalview*mainHSplit1: *n*

Controls the height of the stack trace, stack frame and source panes in the process window. *n* specifies the pixel location of the top of the source pane.

Default: **(window_height/3)**

totalview*mainHSplit2: *n*

Controls the height of the source pane, thread list and action point list in the process window. *n* specifies the pixel location of the top of the thread list and action point list panes.

Default: A function of *window_height*: Tries to give 5 lines in the thread list and action point list panes, and the remainder, at least 20 lines, to the source pane. If it cannot give the source pane at least 20 lines, it shrinks the thread list and action point list panes to zero.

totalview*mainVSplit: *n*

Same as **totalview*mainVSplit1**.

totalview*mainVSplit1: *n*

Controls the location of the partition between the stack trace and stack frame panes in the process window. A value of **–1** centers the partition.

Default: **(window_width/2) – 20**

totalview*mainVSplit2: *n*

Controls the location of the partition between the thread list and action point list panes in the process window. A value of **–1** centers the partition.

Default: **(window_width/2) – 20**

totalview*mainWindLocation: =*width*x*height*+*x*+*y*

Specifies placement of the first main process window.

Default:

width	*height*	*x*	*y*
min(columns(94), screen_width - 5)	**max(456, lines(45))**	**10**	**-150**

totalview*menuArrowForegroundColor: *color*

Sets the menu arrow color to *color*.

Default: **blue** or **green**

totalview*menuCache: {on | **off**}

If off (default), disables menu caching. Not all X servers support menu caching. If your X server doesn't and you have menu caching enabled (**on**), TotalView menus appear blank the second and subsequent times you display them.

totalview*messageStateWindLocation: =*width***x***height*+*x*+*y*

Specifies the placement of the first message state window.

Default:

width	*height*	*x*	*y*
columns(72)	**max(205, lines(15))**	**-80**	**330**

totalview*modulesWindLocation: =*width***x***height*+*x*+*y*

Specifies the placement of the first modules window.

Default:

width	*height*	*x*	*y*
columns(62)	**max(205, lines(15))**	**-75**	**15**

totalview*mouseCursorBackgroundColor: *color*

Sets the mouse cursor background (mask) color to *color*.

Default: **white** or **black**

totalview*mouseCursorForegroundColor: *color*

Sets the mouse cursor foreground (inner) color to *color*.

Default: **red**

totalview*multForegroundColor: *color*

Sets the color of MULT action point signs to *color*.

Default: **purple**

totalview*mutexWindLocation: =*width***x***height*+*x*+*y*

Specifies placement of the first mutex information window.

Default:

width	*height*	*x*	*y*
columns(75)	**lines(15)**	**350**	**300**

totalview*overrideRedirect: {on | **off**}

If off (default), do not create TotalView windows using the **override_redirect** attribute. If on, use the **override_redirect** attribute, which does not give the X window manager a chance to intercept requests.

totalview*ownTitles: {**on** | off}

If on (default), place title bars on TotalView windows. If your window manager is a reparenting one (places its own title bars on windows), turn off this resource.

totalview*popAtBreakpoint: {on | **off**}

If on, sets the **Open (or raise) process window at breakpoint** checkbox to be selected by default. If off (default), sets that checkbox to be deselected by default. See “Handling Signals” on page 48.

Override with:	**–pop_at_breakpoint** option (overrides **off**) **–no_pop_at_breakpoint** option (overrides **on**)

totalview*popOnError: {**on** | off}

If on (default), sets the **Open (or raise) process window on error** checkbox to be selected by default. If off, sets that checkbox to be deselected by default. “Handling Signals” on page 48.

Override with:	**–pop_on_error** option (overrides **off**) **–no_pop_on_error** option (overrides **on**)

totalview*processBarrierStopAll: {**true** | false}

Same as **totalview*processBarrierStopAllRelatedProcessesWhenBreakpoint Hit**.

totalview*processBarrierStopAllRelatedProcessesWhenBreakpointHit: {**true** | false}

If true (default), the default setting for process barrier breakpoints stops all related processes. If false, the default setting for process barrier breakpoints does *not* stop all related processes. See “Process Barrier Breakpoints” on page 201.

totalview*pullRightMenus: {on | **off**}

If off (default), use walking menus. If on, use pull-right menus.

totalview*pvmDebugging: {true | **false**}

If false (default), disables support for debugging the ORNL implementation of Parallel Virtual Machine (PVM) applications. If true, enables support for debugging PVM applications.

Override with:	**–pvm** option (overrides **false**) **–nopvm** option (overrides **true**)

totalview*rootWindLocation: =*width***x***height***+***x***+***y*

Specifies placement of the root window.

Default:

width	*height*	*x*	*y*
min(screen_width - 10, columns(60))	**max(150, lines(12))**	**10**	**10**

totalview*runningFontForegroundColor: *color*

Sets the color of “**R**”, “**S**”, “**M**”, and “**I**” states to *color*.

Default: **green**

totalview*scrollLineSpeed: *n*

Specifies the maximum number of lines per second that TotalView scrolls when you click on arrows at the top and bottom of the scroll bars. To have TotalView scroll as fast as possible, set *n* to **0**.

Default: **40**

totalview*scrollPageSpeed: *n*

Specifies the maximum number of pages per second that TotalView scrolls when you click above or below the elevator box inside the scroll bars. To have TotalView scroll as fast as possible, set *n* to **0**.

Default: **5**

totalview*searchCaseSensitive: {on | **off**}

If off (default), searching for strings is not case-sensitive. If on, searches are case-sensitive.

totalview*searchPath: *dir1*[**,***dir2***,**...]

Specifies a list of directories for the debugger to search when looking for source and object files. This resource serves the same purpose as the **Set Search Directory (d)**command in the process window (see “Setting Search Paths” on page 52). If you use multiple lines, place a backslash (\) at the end of each line, except for the last line.

totalview*serverLaunchEnabled: {**true** | false}

If true (default), TotalView automatically launches the TotalView Debugger Server (**tvdsvr**) when you start to debug a remote process.

totalview*serverLaunchString: *command_string*

Specifies the command string that TotalView uses to automatically launch the TotalView Debugger Server (**tvdsvr**) when you start to debug a remote process. *command_string* is executed by **/bin/sh**. By default, TotalView uses the **rsh** command to start the server, but you can use any other command that can invoke **tvdsvr** on a remote host. If you have no command available for invoking a remote process, you can’t automatically launch the server; therefore, you should set **totalview*serverLaunchEnabled** to **false**.

Default:
rsh %R –n "cd %D && tvdsvr –callback %L –set_pw %P –verbosity %V"

totalview*serverLaunchTimeout: *n*

Specifies the number of seconds that TotalView waits to hear back from the TotalView Debugger Server (**tvdsvr**) that it launched successfully. The number of seconds must be between **1** and **3600** (1 hour).

Default: **30**

totalview*shareActionPoint: {**true** | false}

Same as **totalview*shareActionPointInAllRelatedProcesses**.

totalview*shareActionPointInAllRelatedProcesses: {**true** | false}

If true (default), the default setting for action points will be to share them in all related processes. If false, the default setting for action points will be to *not* share them in all related processes. See “Breakpoints for Multiple Processes” on page 197.

totalview*signalHandlingMode: *action_list*

Modifies the way in which TotalView handles signals. An *action_list* consists of a list of *signal_action* descriptions, separated by spaces:

> *signal_action*[*signal_action*] …

A signal_action description consists of an action, an equal sign (=), and a list of signals:

> *action*=*signal_list*

An *action* can be one of the following: **Error**, **Stop**, **Resend**, or **Discard**. For more information on the meaning of each action, refer to “Handling Signals” on page 48.

A *signal_list* is a list of one or more signal specifiers, separated by commas:

> *signal_specifier*[**,***signal_specifier*] …

A *signal_specifier* can be a signal name (such as **SIGSEGV**), a signal number (such as **11**), or a star (*****), which specifies all signals. We recommend using the signal name rather than the number because number assignments vary across UNIX versions.

The following rules apply when specifying an *action_list*:

- If you specify an action for a signal in an *action_list*, TotalView changes the default action for that signal.
- If you do not specify a signal in the *action_list*, TotalView does not change its default action for the signal.
- If you specify a signal that does not exist for the platform, TotalView ignores it.
- If you specify an action for a signal twice, TotalView uses the last action specified. In other words, TotalView applies the actions from left to right.

If you need to revert the settings for signal handling to TotalView's built-in defaults, use the **Defaults** button in the **Set Signal Handling Mode** dialog box.

For example, to set the default action for the **SIGTERM** signal to **Resend**, you specify the following action list:

"Resend=SIGTERM"

As another example, to set the action for **SIGSEGV** and **SIGBUS** to **Error**, the action for **SIGHUP** and **SIGTERM** to **Resend**, and all remaining signals to **Stop**, you specify the following action list:

"**Stop=* Error=SIGSEGV,SIGBUS Resend=SIGHUP,SIGTERM**"

This action list shows how TotalView applies the actions from left to right. The action list first sets the action for all signals to **Stop**. Then, the action list changes the action for **SIGSEGV** and **SIGBUS** from **Stop** to **Error** and the action for **SIGHUP** and **SIGTERM** from **Stop** to **Resend**.

totalview*sourcePaneTabWidth: *n*

Sets the width of the tab character that is displayed in the source pane. For example, if your source file uses a tab width of 4, set *n* to **4**.

Default: **8**

totalview*spellCorrection: {**verbose** | brief | none}

When you use the **Function or File...** or **Variable...** commands in the process window or edit a type string in a variable window, the debugger checks the spelling of your entries. By default (**verbose**), the debugger displays a dialog box before it corrects spelling. You can set this resource to **brief** to run the spelling corrector silently. (The debugger makes the spelling correction without displaying it in a dialog box first.) You can also set this resource to **none** to disable the spelling corrector.

totalview*stopAll: {**true** | false}

Same as **totalview*stopAllRelatedProcessesWhenBreakpointHit**.

totalview*stopAllRelatedProcessesWhenBreakpointHit: {**true** | false}

If true (default), the default setting for breakpoints will stop all related processes. If false, the default setting for breakpoints will *not* stop all related processes. See “Breakpoints for Multiple Processes” on page 197.

totalview*stopForegroundColor: *color*

Sets the color of STOP and ASM action point signs to *color*.

Default: **red**

totalview*stoppedFontForegroundColor: *color*

Sets the color of “**T**” state to *color*.

Default: **blue** or **yellow2**

totalview*useColor: {**true** | false}

If true (default), enables TotalView use of color. If false, disables all use of color and display using monochrome black on white. This option overrides all other color-related options.

Override with:	**–color** option (overrides **false**)
	–no_color option (overrides **true**)

totalview*userThreads: {**true** | false}

If set to true (default), enables handling of user-level (M:N) thread packages on systems where two-level (kernel and user) thread scheduling is supported. If set to false, disable handling of user-level (M:N) thread packages. Disabling thread support may be useful in situations where you need to debug kernel-level threads, but in most cases, this option is of little use on systems where two-level thread scheduling is used.

Override with:	**–user_threads** option (overrides **false**) **–no_user_threads** option (overrides **true**)

totalview*useTextColor: {**true** | false}

If true (default), enables TotalView use of text color. If false, disables use of text color.

Override with:	**–text_color** option (overrides **false**) **–no_text_color** option (overrides **true**)

totalview*useTitleColor: {**true** | false}

If true (default), enables TotalView use of title color. If false, disables use of title color.

Override with:	**–title_color** option (overrides **false**) **–no_title_color** option (overrides **true**)

totalview*useTransientFor: {**on** | off}

If **off**, use “override redirect” windows, which don’t let you use the window manager to perform operations, such as raise and lower, on dialog boxes. If you use an advanced window manager, you can use the **on** option (default) to specify that the debugger use “transient-for” type windows, which allow you to use the window manager to perform operations on dialog boxes. If you’re using an X11R4 or more recent server and window manager, you should use the **on** option. If you’re using the DECstation’s DEC window manager, you should use the **off** option.

totalview*verbosity: {silent | error | warning | **info**}

Sets the verbosity level of TotalView generated messages.

Default: **info**

totalview*visualizerLaunchString: *command_string*

Specifies the command string that TotalView uses to launch the visualizer when you first visualize something. This is a shell command line, so you can use the shell redirection command to output visualization data-sets to a file (e.g. "**cat >** *your_file*").

Default: **visualize**

totalview*visualizerLaunchEnabled: {**true** | false}

If true (default), TotalView automatically launches the visualizer when you first visualize something. If false, visualization is disabled.

totalview*visualizerMaxRank: *n*

Specifies the default value used in the "**Maximum permissible rank**" field of the Visualizer Launch Window dialog box. This field sets the maximum rank of the array that TotalView will export to the visualizer. TotalView's default visualizer cannot visualize arrays of rank greater than two, however if you are using another visualizer, or just dumping binary data, you can set the limit here.

Default: **2**

totalview*warnStepThrow: {**true** | false}

If set to true (default), and your program throws an exception during a TotalView single-step operation, you will be asked if you wish to stop the single-step operation. The process will be left stopped at the C++ run time library's "throw" routine. If set to false, then TotalView will not catch C++ exception throws during single-step operations, which may cause the single-step operation to lose control of the process, and cause it to run away.

Visualizer X Resources

The TotalView visualizer uses a large number of X resources that are set up in its application defaults file. The X resources documented are a subset of those found in the application defaults file as they are the only ones that may be customized to your preferences. Setting them in your own X resources file overrides the application defaults file.

The default values of the X resources are listed here shown either in a bold typeface in a list of alternative values, or separately if there can be a range of values. They are the settings in the applications defaults file as it is shipped. Your site administrator can edit this file to set the site defaults, therefore your site may have different defaults.

Visualize*data*pick_message.background: *color*

Sets the color of the pick popup window.

Default: **light yellow**

Visualize*directory*auto_visualize.set: {**1** | 0}

Sets the initial state of the auto-visualize option in the directory window. If set (1), when a new data-set is added to the list, it will be visualized automatically using an appropriate method. If cleared (0), the new data-set will not be displayed automatically, and you will have to choose a visualization method for it.

Visualize*directory.width: *width*
Visualize*directory.height: *height*

Sets the initial width and height of the directory window.

Default: *width*=**300**, *height*=**100**

Visualize*graph.width: *width*
Visualize*graph.height: *height*

Sets the initial width and height of the graph data window.

Default: width=**400**, height=**400**

Visualize*graph*lines.set: {**1** | 0}

Sets the initial state of the lines option in the graph window. When set (1), graphs are drawn with lines connecting the data points.

Visualize*graph*points.set: {1 | **0**}

Sets the initial state of the points option in the graph window. When set (1), graphs are drawn with markers on each data point.

Visualize*surface.width: *width*
Visualize*surface.height: *height*

Sets the initial width and height of the surface data window.

Default: *width*=**400**, *height*=**400**

Visualize*surface*mesh.set: {**1** | 0}

Sets the initial state of the mesh option in the surface window. When set (1), the axis grid is projected onto the surface.

Visualize*surface*shade.set: {**1** | 0}

Sets the initial state of the shade option in the surface window. When set (1), the surface is shaded.

Visualize*surface*contour.set: {1 | **0**}

Sets the initial state of the contour option in the surface window. When set (1), contours are displayed on the surface.

Visualize*surface*zone.set: {**1** | 0}

Sets the initial state of the zone option in the surface window. When set (1), the surface is colored according to the value.

Visualize*surface*auto_reduce.set: {**1** | 0}

Sets the initial state of the auto-reduce option in the surface window. When set (1), large data-sets are reduced by averaging to speed display.

Visualize*surface*xrt3dZoneMethod: {**zonecontours** | zonecells}

Specifies how the surface is colored. When set to **zonecontours**, the surface is colored according to its contours. When set to **zonecells**, each cell in the mesh is colored based on the average value in the cell.

Visualize*surface*xrt3dViewNormalized: {1 | **0**}

When set (1), the view of the data-set (before zooming or translation) is maximized to fit the window. Interactive rotation when this resource is set will look “jerky” but will ensure no portion of the display is clipped. When this resource is cleared (0), dynamic rotation will be smooth, but parts of the display (e.g., axes) may be clipped at some viewing angles.

Visualize*surface*xrt3dXMeshFilter: *n*
Visualize*surface*xrt3dYMeshFilter: *n*

Specifies how to display the surface mesh. Every *n*th mesh line will be displayed, where *n* must be an integer greater than or equal to 0. When set to **0**, a value is calculated automatically.

Default: **0**

CHAPTER 12:
TotalView Command Syntax

This chapter summarizes the syntax of the **totalview** command. For the full syntax, use the **man totalview** command to view the online version.

Synopsis

totalview [*filename* [*corefile*]] [*options*]

Description

The TotalView debugger is a source-level debugger with a graphic interface (based on the X Window System) and features for debugging distributed programs, multiprocess programs, and multithreaded programs. You need a workstation or terminal running the X Window System to use TotalView. TotalView is available on a number of different platforms.

Arguments

filename

Specifies the pathname of an executable to be debugged. The name can be an absolute or relative pathname. The executable must be compiled with debugging symbols turned on, normally the **–g** compiler switch. Any multiprocess programs that call fork(), vfork(), or execve() should be linked with the dbfork library.

corefile

Specifies the name of a core file. Specify this argument in addition to *filename* when you want to examine a core file with TotalView:

totalview *filename corefile* [*options*]

Options

If you specify mutually exclusive options (such as **–dynamic** and **–no_dynamic**) on the same command line, the last option listed is used. Some of these options override TotalView X resources described in "X Resources" on page 263. In options that contain underscores (_), you can usually use the option without the underscores. For example, **–nodynamic** is the same as **–no_dynamic**, and **–arrowbgcolor** is the same as **–arrow_bg_color**.

Note: The option, **–***Xresource*=*value*, allows you to set the X resource *Xresource* to *value* from the command line. For example, to set **totalview*stopAll** to **false**, you could specify the command line option **–stopAll=false**. Note that the string "**totalview***" is omitted from the command line. X resource values set from the command line override settings in your X resource file. For a complete list of X resources, see Chapter 11, "X Resources," on page 263.

–a *args* Passes all subsequent arguments (specified by *args*) to the program specified by *filename*. This option must be the *last* one on the command line.

–arrow_bg_color *color*
Sets the background (outline) color of PC arrow to *color*.

Default: **black**

–arrow_color *color* Sets the foreground (inner) color of PC arrow to *color*.

Default: **yellow2**

–background *color* Sets the general background color to *color*.

Default: **white**

–bg *color* Same as **–background**.

–barrier_color *color*
Sets the color of the process barrier breakpoint icon.

Default: blue

–barrier_font_color *color*
Sets the color of the font used to show the **H** and **Hold** indicators for held processes.

Default: blue

–barr_stop_all (Default) Enables process barrier breakpoints to stop all related processes.

–no_barr_stop_all The process barrier breakpoint does *not* stop all related processes.

–break_color *color*
Sets the color of "**B**" state to *color*.

Default: **orange**

–button_bg_color *color*
Sets the button background color to *color*.

Default: background color

–button_fg_color *color*
Sets the button foreground color to *color*.

Default: foreground color

–chase (Default) Displays dialog boxes at the mouse pointer. To display dialog boxes centered in the upper third of the screen, use **–no_chase**.

–no_chase Displays dialog boxes centered in the upper third of the screen.

–color (Default) Enables TotalView use of color.

–no_color Disables all use color, and display using monochrome black on white. This option overrides all other color-related options.

–nc Same as **–no_color**.

–compiler_vars Alpha and SGI only. Show variables created by the Fortran compiler, as well as those in the user's program.

–no_compiler_vars (Default) Do not show variables created by the Fortran compiler.

Some Fortran compilers (Digital f90/f77, SGI 7.2 compilers) output debug information which describes variables that the compiler itself has invented for purposes such as passing the length of character*(*) variables. By default, TotalView suppresses the display of these compiler generated variables.

However you can specify the **–compiler_vars** option or set the **totalview*compilerVars** X resource to true to cause such variables to be displayed. This could be useful if you are looking for a corruption of a run time descriptor or are writing a compiler.

–dbfork (Default) Catches the **fork()**, **vfork()**, and **execve()** system calls if your executable is linked with the **dbfork** library.

–no_dbfork Does not catch **fork()**, **vfork()**, and **execve()** system calls even if your executable is linked with the **dbfork** library.

–debug_file *consoleoutputfile*
Redirects TotalView console output to a file named *consoleoutputfile*.

Default: All TotalView console output is written to **stderr**.

–demangler=*compiler*
Overrides the C++ demangler and mangler TotalView uses by default. Table 30 lists override options.

Table 30. C++ Demangling Command Line Options

Option	**Meaning**
–demangler=cset	IBM xlC C++
–demangler=dec	Digital C++
–demangler=gnu	GNU C++

Table 30. C++ Demangling Command Line Options (Continued)

Option	Meaning
–demangler=irix	SGI IRIX C++
–demangler=kai	KAI KCC C++ 3.2 or greater
–demangler=spro	SunPro C++ 4.0 or greater
–demangler=sun	Sun CFRONT C++
–demangler=usoft	MicroSoft C++

–display *displayname*

Sets the name of the X Windows display to *displayname*. For example, **–display vinnie:0.0** will display TotalView on the machine named “vinnie.”

Default: To the value of the **DISPLAY** environment variable.

–dpvm Digital UNIX only: Enables support for debugging the Digital UNIX implementation of Parallel Virtual Machine (PVM) applications.

–no_dpvm Digital UNIX only: (Default) Disables support for debugging the Digital UNIX implementation of PVM applications.

–dump_core Allows TotalView to dump a core file when it gets an internal error. Useful for debugging TotalView itself.

–no_dump_core (Default) Does not allow TotalView to dump a core file when it gets an internal error.

–dynamic (Default) Loads symbols from shared libraries. This option is available only on platforms that support shared libraries.

–no_dynamic Does not load symbols from shared libraries when reading dynamically linked executables. Setting this option can cause the **dbfork** library to fail because TotalView might not find the **fork()**, **vfork()**, and **execve()** system calls.

–error_color *color*

Sets the color of “**E**”, “**Z**”, and “**?**” states to *color*.

Default: **red**

–eval_color *color*

Sets the color of the EVAL action point signs to *color*.

Default: **orange**

–ext *extension*

Specifies that files with the suffix *extension* are preprocessor input files. TotalView already has built-in extensions for C++ (**.C**, **.cpp**, **.cc**, **.cxx**), Fortran (**.F**), **lex** (**.l**, **.lex**), and **yacc** (**.y**) files.

–font *fontname*

Specifies the font to be used by TotalView.

Default: **fixed**

–fn *fontname*

Same as **–font**.

–foreground *color*

Sets the general foreground color (i.e., the text color) to *color*.

Default: **black**

–fg *color*

Same as **–foreground**.

–global_types

(Default) Specifies that TotalView can assume that type names are globally unique within a program and that all type definitions with the same name are identical. In C++, the standard asserts that this must be true for standard conforming code.

If this option is set, TotalView will attempt to replace an opaque type (**struct foo *p;**) declared in one module, with an identically named defined type (**struct foo { … };**) in a different module.

If TotalView has read the symbols for the module containing the non-opaque type definition, then when displaying variables declared with the opaque type, TotalView will automatically display the variable using the non-opaque type definition.

–no_global_types Specifies that TotalView *cannot* assume that type names are globally unique within a program. You should specify this option if your code has multiple different definitions of the same named type, since otherwise TotalView is likely to pick the wrong definition to substitute for an opaque type.

–grab Forces all keyboard input to go to an open dialog box. Use this option if your window manager uses “click-to-type” mode.

–no_grab (Default) Does not force keyboard input to an open dialog box.

–grab_server (Default) TotalView will grab the X server when posting menus.

–no_grab_server TotalView will not grab the X server when posting menus. Useful for taking screen shots of TotalView’s menus.

–hpf (Default) Enables debugging HPF code at the source level.

–no_hpf Disables debugging HPF source code at the source level.

–hpf_node Enables display of node on which HPF distributed array element resides in the process window.

–no_hpf_node (Default) Disables display of node on which HPF distributed array element resides in the process window.

–ignore_control_c Ignores Control-C and prevents you from terminating the TotalView process from an **xterm** window, which is useful when your program catches the Control-C signal (SIGINT).

–icc Same as **–ignore_control_c**.

–no_ignore_control_c
(Default) Catches Control-C and terminates your TotalView debugging session. To override this, use **–ignore_control_c**.

–nicc Same as **–no_ignore_control_c**.

–iv Turns inverse video on.

–no_iv (Default) Turns inverse video off.

–kcc_classes (Default) Convert structure definitions output by the KCC compiler into classes that show base classes, and virtual base classes in the same way as other C++ compilers. See the description of the X resource "totalview*kccClasses: {true | false}" on page 272 for a description of the conversion performed by TotalView.

–no_kcc_classes Do not convert structure definitions output by the KCC compiler into classes. Virtual bases will show up as pointers, rather than the data.

–lb (Default) Loads action points automatically from the *filename***.TVD.breakpoints** file, providing the file exists.

–nlb Does not load action points automatically from an action points file.

–mc Turns on menu caching. Use this option if your X server supports menu caching. If menus appear blank the second and subsequent times you display them, your X server does not support menu caching.

–nmc (Default) Turns off menu caching.

–menu_arrow_color *color*

Sets the menu arrow color to *color*.

Default: **blue** or **green**

–message_queue (Default) Enable the display of MPI message queues when debugging an MPI program.

–mqd Same as **–message_queue**.

–no_message_queue

Disable the display of MPI message queues when debugging an MPI program. This might be useful if a store corruption is overwriting the message queues and causing TotalView to become confused.

–no_mqd Same as **–no_message_queue**.

–mouse_bg_color *color*
Sets the mouse cursor background (mask) color to *color*.

Default: **white** or **black**

–mouse_fg_color *color*
Sets the mouse cursor foreground (inner) color to *color*.

Default: **red**

–mult_color *color* Sets the color of MULT action point sign to *color*.

Default: **purple**

–parallel (Default) Enable handling of parallel program runtime libraries such as MPI, PE and HPF.

–no_parallel Disable handling of parallel program runtime libraries such as MPI, PE and HPF. This is useful for debugging parallel programs as if they were single process programs.

–pop_at_breakpoint
Sets the **Open (or raise) process window at breakpoint** checkbox to be selected by default. See “Handling Signals” on page 48.

–no_pop_at_breakpoint
(Default) Sets the **Open (or raise) process window at breakpoint** checkbox to be deselected by default. See “Handling Signals” on page 48.

–pop_on_error (Default) Sets the **Open (or raise) process window on error** checkbox to be selected by default. See “Handling Signals” on page 48.

–no_pop_on_error Sets the **Open (or raise) process window on error** checkbox to be deselected by default. See “Handling Signals” on page 48.

–pr Use pull-right menus.

–npr (Default) Use walking menus instead of pull-right menus.

–pvm Enables support for debugging the ORNL implementation of Parallel Virtual Machine (PVM) applications.

–no_pvm (Default) Disables support for debugging the ORNL implementation of PVM applications.

–remote *hostname*[**:***portnumber*]

Debugs an executable that is not running on the same machine as TotalView. For *hostname*, you can specify a TCP/IP hostname, such as **vinnie**, or a TCP/IP address, such as **128.89.0.16**. Optionally, you can specify a TCP/IP port number for *portnumber*, such as **:4174**. When you specify a port number, you disable the auto-launch feature. For more information on the auto-launch feature, see "The Auto-Launch Feature" on page 64.

–r *hostname*[**:***portnumber*]

Same as **–remote**.

–running_color *color*

Sets the color of "**R**", "**S**", "**M**", and "**I**" states to *color*.

Default: **green**

–sb Saves action points automatically to an action points file when you exit TotalView. The file is named *filename*.**TVD**.**breakpoints**.

–nsb (Default) Does not save action points automatically to an action points file when you exit.

–serial *device*[**:***options*]

Debugs an executable that is not running on the same machine as TotalView. For *device*, specify the device name of a serial line, such as **/dev/com1**. Currently, the only *option* you are allowed to specify is the baud rate, which defaults to **38400**. For more information on debugging over a serial line, see "Debugging Over a Serial Line" on page 72.

–signal_handling_mode “*action_list*”
Modifies the way in which TotalView handles signals. You must enclose the *action_list* string in quotation marks to protect it from the shell. Refer to “totalview*signalHandlingMode: action_list” on page 278 for a description of the *action_list* argument.

–shm “*action_list*” Same as **–signal_handling_mode**.

–stop_all (Default) Sets the **Stop All Related Processes when Breakpoint Hit** checkbox to be selected by default. To override this option use **–no_stop_all**. See “Breakpoints for Multiple Processes” on page 197.

–no_stop_all Sets the **Stop All Related Processes when Breakpoint Hit** checkbox to be deselected by default. See “Breakpoints for Multiple Processes” on page 197.

–stop_color *color* Sets the color of STOP and ASM action point signs to *color*.

Default: **red**

–stopped_color *color*
Sets the color of “**T**” state to *color*.

Default: **blue** or **yellow2**

–text_color (Default) Turns text color use on.

–no_text_color Turns text color use off.

–title_color (Default) Turns title color use on.

–tc Same as **–title_color**.

–no_title_color Turns title color use off.

–no_tc Same as **–no_title_color**.

–user_threads (Default) Enable handling of user-level (M:N) thread packages on systems where two-level (kernel and user) thread scheduling is supported.

–no_user_threads Disable handling of user-level (M:N) thread packages. This option may be useful in situations where you need to debug kernel-level threads, but in most cases, this option is of little use on systems where two-level thread scheduling is used.

–verbosity *level* Sets the verbosity level of TotalView generated messages to *level*, which may be one of **silent**, **error**, **warning**, or **info**.

Default: **info**

CHAPTER 13:

TotalView Debugger Server Command Syntax

This chapter summarizes the syntax of the TotalView Debugger Server command, **tvdsvr**, which is used for remote debugging. For more information on remote debugging, refer to "Starting the Debugger Server for Remote Debugging" on page 64.

Synopsis

tvdsvr {**–server** | **–callback** *hostname***:***port* | **–serial** *device*} [*other options*]

Description

The **tvdsvr** debugger server allows TotalView to control and debug a program on a remote machine. To accomplish this, the **tvdsvr** program must run on the remote machine, and it must have access to the executables to be debugged. These executables must have the same absolute pathname as the executable that TotalView is debugging, or the PATH environment variable for **tvdsvr** must include the directories containing the executables.

You must specify either the **–server**, **–callback**, or **–serial** option with the **tvdsvr** command. By default, the TotalView debugger automatically launches **tvdsvr** (known as the auto-launch feature) with the **–callback** option, and the server establishes a connection with TotalView.

If you prefer not to use the auto-launch feature, you can start **tvdsvr** manually and specify the **–server** option. Be sure to make note of the password that **tvdsvr** prints out with the message:

```
pw = hexnumhigh:hexnumlow
```

TotalView will prompt you for *hexnumhigh***:***hexnumlow* later. By default, **tvdsvr** automatically generates a password that is used when establishing connections. If desired, you can use the **–set_pw** option to set a specific password.

To connect to the **tvdsvr** from TotalView, you use the New Program Window and must specify the hostname and TCP/IP port number, *hostname***:***portnumber* on which **tvdsvr** is running. Then, TotalView prompts you for the password for **tvdsvr**.

Options

The following options determine the port number and password necessary for TotalView to connect with **tvdsvr**.

–callback *hostname***:***port*

(Auto-launch feature only) Immediately establishes a connection with the TotalView debugger that is running on *hostname* and listening on *port*, where *hostname* is either a hostname or TCP/IP address. If **tvdsvr** cannot connect with TotalView, it exits. If you specify the **–port**, **–search_port**, and **–server** options with this option, **tvdsvr** ignores them.

–debug_file *consoleoutputfile*

Redirects TotalView Debugger Server console output to a file named *consoleoutputfile*.

Default: All console output is written to **stderr**.

–dpvm

Uses the Digital UNIX implementation of the Parallel Virtual Machine (DPVM) library process as its input channel and registers itself as the DPVM tasker.

Note: This option is not intended for users launching **tvdsvr** manually. When you enable DPVM support within TotalView, TotalView automatically uses this option when it launches **tvdsvr**.

–port *number*

Sets the TCP/IP port number on which **tvdsvr** should communicate with **totalview**. If this TCP/IP port number is busy, **tvdsvr** does not select an alternate port number (that is, it communicates with nothing) unless you also specify **–search_port**.

Default: 4142

–pvm

Uses the ORNL implementation of the Parallel Virtual Machine (PVM) library process as its input channel and registers itself as the ORNL PVM tasker.

Note: This option is not intended for users launching **tvdsvr** manually. When you enable PVM support within TotalView, TotalView automatically uses this option when it launches **tvdsvr**.

–search_port
Searches for an available TCP/IP port number, beginning with the default port (4142) or the port set with the **–port** option and continuing until one is found. When the port number is set, **tvdsvr** displays the chosen port number with the following message:

```
port = number
```

–serial *device*[**:***options*]
Waits for a serial line connection from TotalView. For *device*, specify the device name of a serial line, such as **/dev/com1**. Currently the only *option* you are allowed to specify is the baud rate, which defaults to **38400**. For more information on debugging over a serial line, see "Debugging Over a Serial Line" on page 72.

–server
Listens for and accepts network connections on port 4142 (default). To use a different port, you must specify the **–port** or **–search_port** options. To stop **tvdsvr** from listening and accepting network connections, you must terminate it by pressing Control-C in the terminal window from which it was started or by using the **kill** command.

–set_pw *hexnumhigh***:***hexnumlow*
Sets the password to the 64-bit number specified by the two 32-bit numbers *hexnumhigh* and *hexnumlow*.When a connection is established between **tvdsvr** and TotalView, the 64-bit password passed by TotalView must match the password set with this option. When the password is set, **tvdsvr** displays the selected number in the following message:

```
pw = hexnumhigh:hexnumlow
```

We recommend using this option to avoid connections by other users.

Note: If necessary, you can disable password checking by specifying the **–set_pw 0:0** option with the **tvdsvr** command. Disabling password checking is dangerous; it allows anyone to connect to your server and start programs, including shell commands, using your UID. Therefore, we do not recommend disabling password checking.

–verbosity *level*

Sets the verbosity level of TotalView Debugger Server generated messages to *level*, which may be one of **silent**, **error**, **warning**, or **info**.

Default: **info**

APPENDIX A: Compilers and Environments

This appendix describes the compilers and parallel runtime environments that can be used with this release of TotalView. You must refer to the TotalView release notes included in the TotalView distribution for information on the specific compiler and runtime environment supported by TotalView.

For information on supported operating systems, please refer to Appendix B, “Operating Systems,” on page 321.

This appendix includes:

- Compilers and runtime environments that TotalView supports
- Command line options needed to compile with debugging symbols
- Compiling with exception data on Digital UNIX
- Linking with the **dbfork** library

Supported Compilers and Environments

Please refer to the release notes in your TotalView distribution for the latest information about supported versions of the compilers and parallel runtime environments listed here.

AIX on RS/6000 Systems

Table 31 lists the supported compilers and parallel runtime environments on IBM RS/6000 systems running AIX.

Table 31. Supported Compilers and Environments on AIX

C compilers	• IBM xlc C • FSF GNU C • Cygnus EGCS C
C++ compilers	• IBM xlC C++ • FSF GNU C++ • KAI C++ • Cygnus EGCS C++
Fortran compilers	• IBM xlf for Fortran 77 • IBM xlf90 for Fortran 90
Environments	• Parallel Environment for AIX • MPICH • ORNL PVM • Portland Group HPF

Digital UNIX on Digital Alpha Systems

Table 32 lists the supported compilers and parallel runtime environments on Digital Alpha systems running Digital UNIX.

Table 32. Supported Compilers and Environments on Digital UNIX

C compilers	• Digital UNIX C • FSF GNU C • Cygnus EGCS C
C++ compilers	• Digital UNIX C++ • KAI C++ • FSF GNU C++ • Cygnus EGCS C++
Fortran compilers	• Digital UNIX Fortran 77 • Digital UNIX Fortran 90
Environments	• MPICH • ORNL PVM • Digital DPVM (PVM)

IRIX on SGI MIPS Systems

Table 33 lists the supported compilers and parallel runtime environments on SGI MIPS systems running IRIX.

Table 33. Supported Compilers and Environments on IRIX

C compilers	• SGI MIPSpro C • FSF GNU C • Cygnus EGCS C
C++ compilers	• SGI MIPSpro C++ • FSF GNU C++ • Cygnus EGCS C++
Fortran compilers	• SGI MIPSpro Fortran 77 • SGI MIPSpro Fortran 90
Environments	• MPICH • SGI MPI (part of the Message Passing Toolkit) • ORNL PVM • Portland Group HPF

SunOS 4 on Solaris Systems

Table 34 lists the supported compilers and parallel runtime environments on Solaris 1.x systems running SunOS 4.

Table 34. Supported Compilers and Environments on SunOS 4

C compilers	• SunPro C • Apogee C • FSF GNU C • Cygnus EGCS C
C++ compilers	• SunPro C++ • Apogee C++ • FSF GNU C++ • Cygnus EGCS C++
Fortran compilers	• SunPro Fortran 77
Environments	• MPICH • ORNL PVM

SunOS 5 on SPARC Solaris Systems

Table 35 lists the supported compilers and parallel runtime environments on SPARC Solaris 2.x systems running SunOS 5.

Table 35. Supported Compilers and Environments on SunOS 5 SPARC

C compilers	• SunPro C • WorkShop C • Apogee C • FSF GNU C • Cygnus EGCS C
C++ compilers	• SunPro C++ • WorkShop C++ • KAI C++ • Apogee C++ • FSF GNU C++ • Cygnus EGCS C++
Fortran compilers	• SunPro Fortran 77 • WorkShop Fortran 77 • WorkShop Fortran 90
Environments	• MPICH • ORNL PVM • Portland Group HPF

SunOS 5 on Intel-x86 Solaris Systems

Table 36 lists the supported compilers and parallel runtime environments on SPARC Solaris 2.x systems running SunOS 5.

Table 36. Supported Compilers and Environments on SunOS 5 x86

C compilers	• WorkShop C • FSF GNU C • Cygnus EGCS C
C++ compilers	• WorkShop C++ • FSF GNU C++ • Cygnus EGCS C++
Fortran compilers	• WorkShop Fortran 77
Environments	• MPICH • ORNL PVM

Compiling with Debugging Symbols

You need to compile programs with the **–g** switch and possibly other compiler switches so that debugging symbols are included. This section shows the specific compiler commands to use for each compiler that TotalView supports.

AIX on RS/6000 Systems

Table 37 lists the procedures to compile programs on IBM RS/6000 systems running AIX.

Table 37. Compiling with Debugging Symbols on AIX

Compiler	Compiler Command Line
IBM xlc C	**xlc –g –c** *program*.**c** [1]
FSF GNU C or Cygnus EGCS C	**gcc –g –c** *program*.**c**
IBM xlC C++	**xlC –g –c** *program*.**cxx**
KAI C++	**KCC +K0** [2] **–qnofullpath** [3] **–c** *program*.**cxx**
FSF GNU C++ or Cygnus EGCS C++	**g++ –g –c** *program*.**cxx**
IBM xlf Fortran 77	**xlf –g –c** *program*.**f** [4]
IBM xlf90 Fortran 90	**xlf90 –g –c** *program*.**f90**
Portland Group HPF	**pghpf –g –Mtv –c** *program*.**hpf**

[1] When compiling with any of the IBM xl compilers, if your program will be moved from its creation directory, or you do not want to set the search directory path during debugging, pass the **–qfullpath** switch to the compiler driver. For example: **xlf –qfullpath –g –c** *program*.**f**

[2] When compiling with KCC for debugging, we recommend that you use the **+K0** option and *not* the **–g** option.

3. When compiling with KCC, you must specify the **–qnofullpath** option; KCC is a preprocessor that passes its output to the IBM xlc C compiler that discards **#line** directives necessary for source level debugging if **–qfullpath** is specified.

4. When compiling Fortran programs using the C preprocessor, pass the **–d** switch to the compiler driver. For example: **xlf –d –g –c** *program***.F**

Digital UNIX on Digital Alpha Systems

Table 38 lists the procedures to compile programs on Digital Alpha system running Digital UNIX.

Table 38. Compiling with Debugging Symbols on Digital UNIX

Compiler	Compiler Command Line
Digital UNIX C	**cc –g –c** *program***.c**
FSF GNU C or Cygnus EGCS C	**gcc –g –c** *program***.c**
Digital UNIX C++	**cxx –g –c** *program***.cxx**
KAI C++	**KCC +K0 –c** *program***.cxx** [1]
FSF GNU C++ or Cygnus EGCS C++	**g++ –g –c** *program***.cxx**
Digital UNIX Fortran 77	**f77 –g –c** *program***.f**
Digital UNIX Fortran 90	**f90 –g –c** *program***.f90**

[1] When compiling with KCC for debugging, we recommend that you use the **+K0** option and *not* the **–g** option.

IRIX on SGI MIPS Systems

Table 39 lists the procedures to compile programs on SGI MIPS systems running IRIX.

Table 39. Compiling with Debugging Symbols on IRIX-MIPS

Compiler	**Compiler Command Line** [1]
SGI MIPSpro C	**cc –n32 –g –c** *program*.**c** **cc –64 –g –c** *program*.**c**
FSF GNU C or Cygnus EGCS C	**gcc –g –c** *program*.**c**
SGI MIPSpro C++	**CC –n32 –g –c** *program*.**cxx** **CC –64 –g –c** *program*.**cxx**
KAI C++	**KCC +K0 –c** *program*.**cxx** [2]
FSF GNU C++ or Cygnus EGCS C++	**gcc –g –c** *program*.**cxx**
SGI MIPSpro77	**f77 –n32 –g –c** *program*.**f** **f77 –64 –g –c** *program*.**f**
SGI MIPSpro 90	**f90 –n32 –g –c** *program*.**f90** **f90 –64 –g –c** *program*.**f90**
Portland Group HPF	**pghpf –g –64 –Mtv –c** *program*.**hpf** [3]

[1] Compiling with **–n32** or **–64** is supported. TotalView does not support compiling with **–32**, which is the default for some compilers. You must specify either **–n32** or **–64**.

[2] When compiling with KCC for debugging, we recommend that you use the **+K0** option and *not* the **–g** option.

[3] You must compiler your programs with the **pghpf –64** compiler option; on SGI IRIX, TotalView can debug 64-bit executables only.

SunOS 4 on Solaris Systems

Table 40 lists the procedures to compile programs on SunOS 4 Solaris 1.x systems running SunOS 4.

Table 40. Compiling with Debugging Symbols on SunOS 4

Compiler	Compiler Command Line
SunPro C	**cc –g –c** *program*.**c**
Apogee C	**apcc –g –c** *program*.**c**
FSF GNU C or Cygnus EGCS C	**gcc –g –c** *program*.**c**
SunPro C++	**CC –g –c** *program*.**cxx**
Apogee C++	**apCC –g –c** *program*.**cxx**
FSF GNU C++ or Cygnus EGCS C++	**g++ –g –c** *program*.**cxx**
SunPro Fortran 77	**f77 –g –c** *program*.**f**

SunOS 5 on SPARC or Intel-x86 Solaris Systems

Table 41 lists the procedures to compile programs on SunOS 5 SPARC or Intel-x86.

Table 41. Compiling with Debugging Symbols on SunOS 5

Compiler [1]	**Compiler Command Line**
SunPro C or WorkShop C	**cc –g –c** *program***.c**
Apogee C	**apcc –g –c** *program***.c**
FSF GNU C or Cygnus EGCS C	**gcc –g –c** *program***.c**
SunPro C++ or WorkShop C	**CC –g –c** *program***.cxx**
Apogee C++	**ap**CC **–g –c** *program***.cxx**
FSF GNU C++ or Cygnus EGCS C++	**g++ –g –c** *program***.cxx**
KAI C++ [2]	**KCC +K0 –c** *program***.cxx** [3]
SunPro Fortran 77 or WorkShop Fortran 77	**f77 –g –c** *program***.f**
WorkShop Fortran 90	**f90 –g –c** *program***.f90**
Portland Group HPF [4]	**pghpf –g –Mtv –c** *program*.**hpf**

[1] On SunOS 5 Intel-x86 Solaris systems, TotalView supports only the WorkShop C, C++, and f77 compilers, and the GNU or EGCS C and C++ compilers.

[2] KCC is supported on SunOS 5 SPARC only.

[3] When compiling with KCC for debugging, we recommend that you use the **+K0** option and *not* the **–g** option.

[4] PGHPF is supported on SunOS 5 SPARC only.

Compiling with Exception Data on Alpha Digital UNIX

If you receive the following error message when you load an executable into TotalView, you may need to compile your program so that exception data is included:

> “Cannot find exception information. Stack backtraces may not be correct.”

To provide a complete stack backtrace in all situations, TotalView needs the exception data to be included in the compiled executable. To compile with exception data, you need to use the following switches:

% **cc –Wl,–u,_fpdata_size** *program***.c**

Where:

–Wl Passes the arguments that follow to another compilation phase (**–W**), which in this case is the linker (**l**). Each argument is separated by a comma (,).

,–u Causes the linker to mark the next argument (**_fpdata_size**) as undefined.

,_fpdata_size Marks the **_fpdata_size** variable as undefined, which forces the exception data into the executable.

Compiling with exception data increases the size of your executable slightly. If you choose not to compile with exception data, TotalView can provide correct stack backtraces in most situations, but not in all situations.

Linking with the dbfork Library

If your program uses the **fork()** and **execve()** system calls, and you want to debug the child processes, you need to link programs with the **dbfork** library.

AIX on RS/6000 Systems

Add one of the following arguments to the command that you use to link your programs:

- **/usr/totalview/lib/libdbfork.a**
- **–L/usr/totalview/lib –ldbfork**

  /usr/totalview/lib/libdbfork.a –bkeepfile:/usr/totalview/lib/libdbfork.a

  –L/usr/totalview/lib –ldbfork –bkeepfile:/usr/totalview/lib/libdbfork.a

For example:

```
% cc –o program program.c –L/usr/totalview/lib –ldbfork
–bkeepfile:/usr/totalview/lib/libdbfork.a
```

When you use **gcc** or **g++**, use the **–Wl,–bkeepfile** option instead of **–bkeepfile**, which will pass the same option to the binder. For example:

```
% cc –o program program.c –L/usr/totalview/lib –ldbfork
–Wl,–bkeepfile:/usr/totalview/lib/libdbfork.a
```

Linking C++ Programs with dbfork

The binder option **–bkeepfile** currently cannot be used with the IBM xlC C++ compiler. The compiler passes all binder options to an additional pass called **munch**, which cannot handle the **–bkeepfile** option.

To work around this problem, we have provided the C++ header file **libdbfork.h**. You must include this file somewhere in your C++ program, in order to force the components of the dbfork library to be kept in your executable. The file **libdbfork.h** is included only with TotalView for the RS/6000 platform, so the include should be placed under a **#ifdef _AIX**. For example:

```
#ifdef _AIX
#include “/usr/totalview/lib/libdbfork.h”
#endif
int main (int argc, char *argv[])
{
}
```

Alpha Digital UNIX

Add one of the following arguments to the command that you use to link your programs:

/opt/totalview/lib/libdbfork.a

–L/opt/totalview/lib –ldbfork.a

For example:

% **cc –o** *program program*.**c –L/opt/totalview/lib –ldbfork**

As an alternative, you can set the LD_LIBRARY_PATH environment variable and omit the **–L** option on the command line:

setenv LD_LIBRARY_PATH /opt/totalview/lib

SunOS 4

Add one of the following arguments to the command that you use to link your programs:

/usr/totalview/lib/libdbfork.a

–L/usr/totalview/lib –ldbfork.a

For example:

% **cc –o** *program program*.**c –L/usr/totalview/lib –ldbfork**

As an alternative, you can set the LD_LIBRARY_PATH environment variable and omit the **–L** option on the command line:

setenv LD_LIBRARY_PATH /usr/totalview/lib

SunOS 5 SPARC or Intel-x86

Add one of the following arguments to the command that you use to link your programs:

/opt/totalview/lib/libdbfork.a

–L/opt/totalview/lib –ldbfork.a

For example:

% **cc –o** *program program***.c –L/opt/totalview/lib –ldbfork**

As an alternative, you can set the LD_LIBRARY_PATH environment variable and omit the **–L** option on the command line:

setenv LD_LIBRARY_PATH /opt/totalview/lib

IRIX6-MIPS

Add one of the following arguments to the command that you use to link your programs.

If you are compiling your code with **–n32**, use the following arguments:

/opt/totalview/lib/libdbfork_n32.a

–L/opt/totalview/lib –ldbfork_n32.a

For example:

% **cc –n32 –o** *program program***.c –L/opt/totalview/lib –ldbfork_n32**

If you are compiling your code with **–64**, use the following arguments:

/opt/totalview/lib/libdbfork.a_n64.a

–L/opt/totalview/lib –ldbfork_n64.a

For example:

% **cc –64 –o** *program program***.c –L/opt/totalview/lib –ldbfork_n64**

As an alternative, you can set the LD_LIBRARY_PATH environment variable and omit the **–L** option on the command line:

setenv LD_LIBRARY_PATH /opt/totalview/lib

APPENDIX B: Operating Systems

This appendix describes the operating system features that can be used with TotalView. This appendix includes the following topics:

- Supported versions
- Mounting the **/proc** file system (Digital UNIX, IRIX, and SunOS 5 only)
- Swap space
- Shared libraries
- Remapping keys (Sun keyboards only)
- Capabilities and characteristics
- Expression system support

Supported Operating Systems

For a complete list of hardware and software requirements including required OS patches and restrictions, see the TotalView release notes in your software distribution. This version of TotalView supports the following operating system versions:

- Digital Alpha workstations running Digital UNIX versions V4.0, V4.0A, V4.0B, V4.0C and V4.0D. All versions *require patches* See “Digital UNIX Patch Procedures” in the TotalView Release Notes for instructions.
- IBM RS/6000 and SP systems running AIX versions 4.1, 4.2, 4.3, or 4.3.1
- Sun Sparc SunOS 4 (Solaris 1.x) systems running SunOS versions 4.1.1, 4.1.2, 4.1.3, or 4.1.4
- Sun Sparc SunOS 5 (Solaris 2.x) systems running SunOS versions 5.5, 5.5.1, or 5.6. (Solaris 2.5, 2.5.1, or 2.6)
- Intel-x86 SunOS 5 (Solaris 2.x) systems running SunOS versions 5.6. (Solaris 2.6)
- SGI IRIX 6.2, 6.3, 6.4, or 6.5 on any MIPS R4000, R4400, R4600, R5000, R8000, or R10000 processor-based systems
- QSW CS-2 based on Sparc Solaris 2.5.1 or 2.6

Note: QSW CS-2 TotalView is nearly identical to TotalView on Sun Solaris 2.*x* systems.

Please see the *TotalView Supplement for CS-2 Users* for more information of CS-2 TotalView specific features.

Mounting the /proc File System

Digital UNIX, SunOS 5, and IRIX

To debug programs on Digital UNIX, SunOS 5, and IRIX with TotalView, you need to mount the **/proc** file system.

If you receive one of the following errors from TotalView, the **/proc** file system might not be mounted:

- job_t::launch, creating process: process not found
- Error launching process while trying to read dynamic symbols
- Creating Process... Process not found
  Clearing Thrown Flag
  Operation Attempted on an unbound d_process object.

To determine whether the **/proc** file system is mounted, enter the appropriate command from Table 42.

Table 42. Commands for Determining Whether /**proc** is Mounted

Operating System	Command
Digital UNIX	% **/sbin/mount –t procfs** `/proc on /proc type procfs (rw)`
SunOS 5	% **/sbin/mount | grep /proc** `/proc on /proc read/write/setuid on Thu Jun 9 18:2208 1994`
IRIX	% **/sbin/mount | grep /proc** `/proc on /proc type proc (rw)`

If you receive the message shown from the mount command, the **/proc** file system is mounted.

Digital UNIX and SunOS 5

To make sure that the **/proc** file system is mounted each time your system boots, add the appropriate line from Table 43 to the appropriate file. Then, to mount the **/proc** file system, enter the following command:

```
% /sbin/mount /proc
```

Table 43. Commands for Automatically Mounting **/proc** File System

Operating System	Name of File	Line to add
Digital UNIX	**/etc/fstab**	`/proc    /proc  procfs rw 0 0`
SunOS 5	**/etc/vfstab**	`/proc  -  /proc  proc  -  no  -`

IRIX

To make sure that the **/proc** file system is mounted each time your system boots, make sure that **/etc/rc2** issues the **/etc/mntproc** command. Then, to mount the /proc file system, enter the following command:

```
% /etc/mntproc
```

Swap Space

Debugging large programs can exhaust the swap space on your machine. If you run out of swap space, TotalView exits with a fatal error, such as:

- Fatal Error: Out of space trying to allocate

  This error indicates that either:

  - TotalView failed to allocate dynamic memory. It can occur anytime during a TotalView session.
  - The data size limit in the C shell is too small. You can use the C shell's **limit** command to increase the data size limit. For example:

  ```
  % limit datasize unlimited
  ```

- job_t::launch, creating process: Operation failed

  This error indicates that the **fork()** or **execve()** system call failed while TotalView was creating a process to debug. It can happen when TotalView tries to create a process.

Digital UNIX

To find out how much swap space has been allocated and is currently being used, use the **swapon** command on Digital UNIX:

```
% /sbin/swapon –s

Total swap allocation:
 Allocated space: 85170 pages (665MB)
 Reserved space: 14216 pages ( 16%)
 Available space: 70954 pages ( 83%)

Swap partition /dev/rz3b:
 Allocated space: 16384 pages (128MB)
 In-use space: 2610 pages ( 15%)
 Free space: 13774 pages ( 84%)

Swap partition /dev/rz3h:
 Allocated space: 52402 pages (409MB)
 In-use space: 2575 pages ( 4%)
 Free space: 49827 pages ( 95%)

Swap partition /dev/rz1b:
 Allocated space: 16384 pages (128MB)
 In-use space: 2592 pages ( 15%)
 Free space: 13792 pages ( 84%)
```

In this example, 665MB of swap has been allocated, and 106MB of it is currently in use.

To find out how much swap space is in use while you are running TotalView:

```
% /bin/ps –o LFMT
```

For example, in this case the value in the VSZ column is 4.45MB:

```
UID   PID  PPID  CP  PRI  NI  VSZ     RSS  WCHAN S TT  TIME     COMMAND
12270 5340 5293  0   41   0   4.45M   1.27 event S p0  0:00.17  totalview a.out
```

To add swap space, use the **/sbin/swapon(8)** command. You must be **root** to use this command. For more information, refer to the on-line manual page for this command.

AIX

To find out how much swap space has been allocated and is currently being used, use the **pstat** command:

```
% /usr/sbin/pstat –s

PAGE SPACE:

          USED PAGES   FREE PAGES
             7555        115325
```

In this example, 122880 (7555 + 115325) pages of swap have been allocated. 7555 pages are currently in use and 115325 pages are free.

To find out how much swap space is in use while you are running TotalView:

1. Start TotalView with a large executable:

   % **totalview** *executable*

2. Press Control-Z to suspend TotalView.
3. Use the following command to see how much swap space TotalView is using:

   % **ps u**

   For example, in this case the value in the SZ column is 5476KB:

```
USER    PID %CPU %MEM   SZ  RSS    TTY STAT    STIME  TIME COMMAND
smith 15080  0.0  6.0 5476 5476  pts/1 T    09:31:43  0:00 totalview executable
```

To add swap space, use the AIX system management tool, **smit**. Use the following path through the **smit** menus:

System Storage Management »Logical Volume Manager »Paging Space

SunOS 4

To find out how much swap space has been allocated and is currently being used, use the **pstat** command:

```
% /etc/pstat –T
136/582 files
  2/ 26 inodes
 38/138 processes
5872/157896 swap
```

In this example, 157896K of swap has been allocated, and 5872K of it is currently in use.

To find out how much swap space is in use while you are running TotalView:

1. Start TotalView with a large executable:

   ```
   % totalview executable
   ```

2. Press Control-Z to suspend TotalView.
3. Use the following command to see how much swap space TotalView is using:

   ```
   %  ps u
   ```

   For example, in this case the value in the SZ column is 66043K, or 66MB:

```
USER    PID   %CPU %MEM  SZ RSS  TT  STAT START   TIME  COMMAND
smith   13276  3.5 17.9660439844 pf  S    15:40   0:51  totalview executable
```

To add swap space, use the **mkfile(8)** and **swapon(8)** commands. You must be **root** to use these commands. For more information, refer to the online manual pages for these commands.

SunOS 5

To find out how much swap space has been allocated and is currently being used, use the **swap** command:

```
%  /usr/sbin/swap –s
total: 16192K bytes allocated + 7140K bytes
reserved = 23332K used, 63456K available
```

To find out how much swap space is in use while you are running TotalView:

1. Start TotalView with a large executable:

   ```
   %  totalview executable
   ```

2. Press Control-Z to suspend TotalView.
3. Use the following command to see how much swap space TotalView is using:

   ```
   %  /bin/ps –l
   ```

For example, in this case the value in the SZ column is 1036 pages, with each page being 4K in size.

```
F S   UID   PID   PPID  C  PRI NI     ADDR   SZ   WCHAN  TTY   TIME COMD
8 T 14694   3456  2558  80   1 20  ff451000  1036       pts/4  0:01 totalview
```

To add swap space, use the **mkfile(1M)** and **swap(1M)** commands. You must be root to use these commands. For more information, refer to the on-line manual pages for these commands.

IRIX

To find out how much swap space has been allocated and is currently being used, use the **swap** command:

```
% /sbin/swap –s
total: 1.55m allocated + 124.47m add'l reserved =
126.02m bytes used, 250.94m bytes available
```

To find out how much swap space is in use while you are running TotalView:

1. Start TotalView with a large executable:

   % **totalview** *executable*

2. Press Control-Z to suspend TotalView.
3. Use the following command to see how much swap space TotalView is using:

   % **/bin/ps –l**

For example, in this case the value in the SZ column is 584 pages.

```
  F S   UID   PID  PPID  C  PRI NI  P   SZ:RSS   WCHAN TTY     TIME CMD
 b0 T 14694 26236 26271  5   62 20  *  584:373       - ttyq6   0:01 totalview
```

Use the following command to determine the number of bytes in a page:

% **sysconf PAGESIZE**

To add swap space, use the **mkfile(1M)** and **swap(1M)** commands. You must be root to use these commands. For more information, refer to the on-line manual pages for these commands.

Shared Libraries

TotalView supports dynamically linked executables, that is, executables that are linked with shared libraries.

When you start TotalView with a dynamically linked executable, TotalView loads an additional set of symbols for the shared library, as indicated in the shell from which you started TotalView. To accomplish this, TotalView:

- Runs a sample process and discards it.
- Reads information from the process.
- Reads the symbol table for each library.

Note: TotalView does not read the symbol table of shared libraries that are loaded at runtime using the following functions:

For Digital UNIX, SunOS 4, SunOS 5, and IRIX: **dlopen()** function

For AIX: **dlopen()** or **load()** functions.

When you create a process without starting it, and the process does not include shared libraries, the program counter points to the entry point of the process, the **start** routine. If the process does include shared libraries, however, TotalView takes the following actions:

- Runs the dynamic loader (SunOS 4 and SunOS 5: **ld.so**, Digital UNIX**: /sbin/loader**, IRIX: **rld**).
- Sets the PC to point to the location after the invocation of the dynamic loader but before the invocation of the **main** routine.

When you attach to a process that uses shared libraries, TotalView takes the following actions:

- If you attached to the process after the dynamic loader ran, then TotalView loads the dynamic symbols for the shared library.
- If you attached to the process before it runs the dynamic loader, TotalView allows the process to run the dynamic loader to completion. Then, TotalView loads the dynamic symbols for the shared library.

If desired, you can suppress the use of shared libraries by starting TotalView with the **–no_dynamic** option. Refer to Chapter 12, “TotalView Command Syntax,” on page 287 for details on this TotalView start-up option.

If you believe that a shared library has changed since you started a Totalview session, you can use the **Reload Shared Library Information** command on the **Current/Update/Relative** submenu to reload library symbol tables. Be aware that only some systems such as AIX permit you to reload library information.

Remapping Keys

On the SunOS 4 and SunOS 5 keyboards, you may need to remap the page-up and page-down keys to the Prior and Next keysym so that you can scroll TotalView windows with the page-up and page-down keys. To do so, add the following lines to your X Window System start-up file:

```
# Remap F29/F35 to PgUp/PgDn
xmodmap -e 'keysym F29 = Prior'
xmodmap -e 'keysym F35 = Next'
```

Expression System

Depending on the target platform, TotalView supports:

- An interpreted expression system only
- Both an interpreted and a compiled expression system

Unless stated otherwise below, TotalView supports interpreted expressions only. See “Interpreted vs. Compiled Expressions” on page 209 for more information on the differences between interpreted and compiled expressions.

AIX

On AIX, TotalView supports compiled and interpreted expressions. TotalView also supports assembler in expressions.

Digital UNIX

On Digital Unix, TotalView supports compiled and interpreted expressions. TotalView also supports Assembler in expressions.

Expression on the Power

Some program functions called from the TotalView expression system on the Power architecture cannot have floating-point arguments which are passed by value. However, in functions with a variable number of arguments, floating-point arguments *can* be in the varying part of the argument list. For example, you can include floating-point arguments with calls to **printf**:

```
double d = 3.14159;
printf("d = %f\n", d);
```

APPENDIX C: Architectures

This appendix describes the architectures TotalView supports, including:

- Power
- Alpha
- SPARC
- MIPS
- Intel-x86 (Intel 80386, 80486 and Pentium processors)

It includes the following topics for each architecture:

- General registers
- Floating-point registers
- Floating-point format

Power

Power General Registers

TotalView displays Power general registers in the stack frame pane of the process window. Table 44 describes how TotalView treats each general register, and the actions you can take with each register.

Table 44. Power General Purpose Integer Registers

Register	Description	Data Type	Edit	Dive	Specify in Expression
R0	General register 0	**<int>**	yes	yes	**$r0**
SP	Stack pointer	**<int>**	yes	yes	**$sp**
RTOC	TOC pointer	**<int>**	yes	yes	**$rtoc**
R3 – R31	General registers 3 – 31	**<int>**	yes	yes	**$r3 – $r31**
INUM		**<int>**	yes	no	**$inum**
PC	Program counter	**<code>[]**	no	yes	**$pc**
SRR1	Machine status save/restore register	**<int>**	yes	no	**$srr1**
LR	Link register	**<int>**	yes	no	**$lr**
CTR	Counter register	**<int>**	yes	no	**$ctr**
CR	Condition register	**<int>**	yes	no	**$cr**
XER	Integer exception register	**<int>**	yes	no	**$xer**
DAR	Data address register	**<int>**	yes	no	**$dar**
MQ	MQ register	**<int>**	yes	no	**$mq**
MSR	Machine state register	**<int>**	yes	no	**$msr**
SEG0 – SEG9	Segment registers 0 – 9	**<int>**	yes	no	**$seg0 – $seg9**

Table 44. Power General Purpose Integer Registers (Continued)

Register	Description	Data Type	Edit	Dive	Specify in Expression
SG10 – SG15	Segment registers 10 –15	**<int>**	yes	no	**$sg10 – $sg15**
SCNT	SS_COUNT	**<int>**	yes	no	**$scnt**
SAD1	SS_ADDR 1	**<int>**	yes	no	**$sad1**
SAD2	SS_ADDR 2	**<int>**	yes	no	**$sad2**
SCD1	SS_CODE 1	**<int>**	yes	no	**$scd1**
SCD2	SS_CODE 2	**<int>**	yes	no	**$scd2**
TID		**<int>**	yes	no	

Power MSR Register

For your convenience, TotalView interprets the bit settings of the Power MSR register. You can edit the value of the MSR and set it to any of the bit settings outlined in Table 45.

Table 45. Power MSR Register Bit Settings

Value	Bit Setting	Meaning
0x00040000	POW	Power management enable
0x00020000	TGPR	Temporary GPR mapping
0x00010000	ILE	Exception little-endian mode
0x00008000	EE	External interrupt enable
0x00004000	PR	Privilege level
0x00002000	FP	Floating-point available
0x00001000	ME	Machine check enable
0x00000800	FE0	Floating-point exception mode 0

Table 45. Power MSR Register Bit Settings (Continued)

Value	Bit Setting	Meaning
0x00000400	SE	Single-step trace enable
0x00000200	BE	Branch trace enable
0x00000100	FE1	Floating-point exception mode 1
0x00000040	IP	Exception prefix
0x00000020	IR	Instruction address translation
0x00000010	DR	Data address translation
0x00000002	RI	Recoverable exception
0x00000001	LE	Little-endian mode enable

Power Floating-Point Registers

TotalView displays the Power floating-point registers in the stack frame pane of the process window. Table 46 describes how TotalView treats each floating-point register, and the actions you can take with each register.

Table 46. Power Floating-Point Registers

Register	Description	Data Type	Edit	Dive	Specify in Expression
F0 – F31	Floating-point registers 0 – 31	**<double>**	yes	yes	**$f0 – $f31**
FPSCR	Floating-point status register	**<int>**	yes	no	**$fpscr**
FPSCR2	Floating-point status register 2	**<int>**	yes	no	**$fpscr2**

Power FPSCR Register

For your convenience, TotalView interprets the bit settings of the Power FPSCR register. You can edit the value of the FPSCR and set it to any of the bit settings outlined in Table 47.

Table 47. Power FPSCR Register Bit Settings

Value	Bit Setting	Meaning
0x80000000	FX	Floating-point exception summary
0x40000000	FEX	Floating-point enabled exception summary
0x20000000	VX	Floating-point invalid operation exception summary
0x10000000	OX	Floating-point overflow exception
0x08000000	UX	Floating-point underflow exception
0x04000000	ZX	Floating-point zero divide exception
0x02000000	XX	Floating-point inexact exception
0x01000000	VXSNAN	Floating-point invalid operation exception for SNaN
0x00800000	VXISI	Floating-point invalid operation exception: $\infty - \infty$
0x00400000	VXIDI	Floating-point invalid operation exception: ∞ / ∞
0x00200000	VXZDZ	Floating-point invalid operation exception: 0 / 0
0x00100000	VXIMZ	Floating-point invalid operation exception: $\infty * \infty$
0x00080000	VXVC	Floating-point invalid operation exception: invalid compare
0x00040000	FR	Floating-point fraction rounded
0x00020000	FI	Floating-point fraction inexact
0x00010000	FPRF=(C)	Floating-point result class descriptor
0x00008000	FPRF=(L)	Floating-point less than or negative
0x00004000	FPRF=(G)	Floating-point greater than or positive

Table 47. Power FPSCR Register Bit Settings (Continued)

Value	Bit Setting	Meaning
0x00002000	FPRF=(E)	Floating-point equal or zero
0x00001000	FPRF=(U)	Floating-point unordered or NaN
0x00011000	FPRF=(QNAN)	Quiet NaN; alias for FPRF=(C+U)
0x00009000	FPRF=(-INF)	-Infinity; alias for FPRF=(L+U)
0x00008000	FPRF=(-NORM)	-Normalized number; alias for FPRF=(L)
0x00018000	FPRF=(-DENORM)	-Denormalized number; alias for FPRF=(C+L)
0x00012000	FPRF=(-ZERO)	-Zero; alias for FPRF=(C+E)
0x00002000	FPRF=(+ZERO)	+Zero; alias for FPRF=(E)
0x00014000	FPRF=(+DENORM)	+Denormalized number; alias for FPRF=(C+G)
0x00004000	FPRF=(+NORM)	+Normalized number; alias for FPRF=(G)
0x00005000	FPRF=(+INF)	+Infinity; alias for FPRF=(G+U)
0x00000400	VXSOFT	Floating-point invalid operation exception: software request
0x00000200	VXSQRT	Floating-point invalid operation exception: square root
0x00000100	VXCVI	Floating-point invalid operation exception: invalid integer convert
0x00000080	VE	Floating-point invalid operation exception enable
0x00000040	OE	Floating-point overflow exception enable
0x00000020	UE	Floating-point underflow exception enable
0x00000010	ZE	Floating-point zero divide exception enable
0x00000008	XE	Floating-point inexact exception enable

Table 47. Power FPSCR Register Bit Settings (Continued)

Value	Bit Setting	Meaning
0x00000004	NI	Floating-point non-IEEE mode enable
0x00000000	RN=NEAR	Round to nearest
0x00000001	RN=ZERO	Round toward zero
0x00000002	RN=PINF	Round toward +infinity
0x00000003	RN=NINF	Round toward -infinity

Using the Power FPSCR Register

On AIX, if you compile your program to catch floating point exceptions (IBM compiler **-qflttrap** option), you can change the value of the FPSCR within TotalView to customize the exception handling for your program.

For example, if your program inadvertently divides by zero, you can edit the bit setting of the FPSCR register in the stack frame pane. In this case, you would change the bit setting for the FPSCR to include 0x10 (as shown in Table 47) so that TotalView traps the "divide by zero" exception. The string displayed next to the FPSR register should now include "**ZE**". Now, when your program divides by zero, it receives a SIGTRAP signal, which will be caught by TotalView. See Chapter 3, "Setting Up a Debugging Session," on page 35 and "Handling Signals" on page 48 for more information. If you did not set the bit for trapping divide by zero or you did not compile to catch floating point exceptions, your program would not stop and the processor would set the "**ZX**" bit.

Power Floating-Point Format

The Power architecture supports the IEEE floating-point format.

SPARC

SPARC General Registers

TotalView displays the SPARC general registers in the stack frame pane of the process window. Table 48 describes how TotalView treats each general register, and the actions you can take with each register.

Table 48. SPARC General Registers

Register	Description	Data Type	Edit	Dive	Specify in Expression
G0	Global zero register	**<int>**	no	no	**$g0**
G1 – G7	Global registers	**<int>**	yes	yes	**$g1 – $g7**
O0 – O5	Outgoing parameter registers	**<int>**	yes	yes	**$o0 – $o5**
SP	Stack pointer	**<int>**	yes	yes	**$sp**
O7	Temporary register	**<int>**	yes	yes	**$o7**
L0 – L7	Local registers	**<int>**	yes	yes	**$l0 – $l7**
I0 – I5	Incoming parameter registers	**<int>**	yes	yes	**$i0 – $i5**
FP	Frame pointer	**<int>**	yes	yes	**$fp**
I7	Return address	**<int>**	yes	yes	**$i7**
PSR	Processor status register	**<int>**	yes	no	**$psr**
Y	Y register	**<int>**	yes	yes	**$y**
WIM	WIM register	**<int>**	no	no	
TBR	TBR register	**<int>**	no	no	
PC	Program counter	**<code>[]**	no	yes	**$pc**
nPC	Next program counter	**<code>[]**	no	yes	**$npc**

SPARC PSR Register

For your convenience, TotalView interprets the bit settings of the SPARC PSR register. You can edit the value of the PSR and set some of the bits outlined in Table 49.

Table 49. SPARC PSR Register Bit Settings

Value	Bit Setting	Meaning
ET	0x00000020	Traps enabled
PS	0x00000040	Previous supervisor
S	0x00000080	Supervisor mode
EF	0x00001000	Floating-point unit enabled
EC	0x00002000	Coprocessor enabled
C	0x00100000	Carry condition code
V	0x00200000	Overflow condition code
Z	0x00400000	Zero condition code
N	0x00800000	Negative condition code

SPARC Floating-Point Registers

TotalView displays the SPARC floating-point registers in the stack frame pane of the process window. Table 50 describes how TotalView treats each floating-point register, and the actions you can take with each register.

Table 50. SPARC Floating-Point Registers

Register	Description	Data Type	Edit	Dive	Specify in Expression
F0 – F31	Floating-point registers (*f* registers), used singly[1]	**<float>**	yes	yes	**$f0 – $f31**
F0/F1 – F30/F31	Floating point registers (*f* registers), used as pairs[1]	**<double>**	yes	yes	**$f0_f1 – $f30_f31**

Table 50. SPARC Floating-Point Registers (Continued)

Register	Description	Data Type	Edit	Dive	Specify in Expression
FPCR	Floating-point control register	**<int>**	no	no	**$fpcr**
FPSR	Floating-point status register	**<int>**	yes	no	**$fpsr**

[1] TotalView allows you to use these registers singly or in pairs, depending on how they are used by your program. For example, if you use F1 by itself, its type is **<float>**, but if you use the F0/F1 pair, its type is **<double>**.

SPARC FPSR Register

For your convenience, TotalView interprets the bit settings of the SPARC FPSR register. You can edit the value of the FPSR and set it to any of the bit settings outlined in Table 51.

Table 51. SPARC FPSR Register Bit Settings

Value	Bit Setting	Meaning
CEXC=NX	0x00000001	Current inexact exception
CEXC=DZ	0x00000002	Current divide by zero exception
CEXC=UF	0x00000004	Current underflow exception
CEXC=OF	0x00000008	Current overflow exception
CEXC=NV	0x00000010	Current invalid exception
AEXC=NX	0x00000020	Accrued inexact exception
AEXC=DZ	0x00000040	Accrued divide by zero exception
AEXC=UF	0x00000080	Accrued underflow exception
AEXC=OF	0x00000100	Accrued overflow exception
AEXC=NV	0x00000200	Accrued invalid exception
EQ	0x00000000	Floating-point condition =

Table 51. SPARC FPSR Register Bit Settings (Continued)

Value	Bit Setting	Meaning
LT	0x00000400	Floating-point condition <
GT	0x00000800	Floating-point condition >
UN	0x00000c00	Floating-point condition unordered
QNE	0x00002000	Queue not empty
NONE	0x00000000	Floating-point trap type None
IEEE	0x00004000	Floating-point trap type IEEE Exception
UFIN	0x00008000	Floating-point trap type Unfinished FPop
UIMP	0x0000c000	Floating-point trap type Unimplemented FPop
SEQE	0x00010000	Floating-point trap type Sequence Error
NS	0x00400000	Non-standard floating-point FAST mode
TEM=NX	0x00800000	Trap enable mask – Inexact Trap Mask
TEM=DZ	0x01000000	Trap enable mask – Divide by Zero Trap Mask
TEM=UF	0x02000000	Trap enable mask – Underflow Trap Mask
TEM=OF	0x04000000	Trap enable mask – Overflow Trap Mask
TEM=NV	0x08000000	Trap enable mask – Invalid Operation Trap Mask
EXT	0x00000000	Extended rounding precision – Extended precision
SGL	0x10000000	Extended rounding precision – Single precision
DBL	0x20000000	Extended rounding precision – Double precision
NEAR	0x00000000	Rounding direction – Round to nearest (tie-even)
ZERO	0x40000000	Rounding direction – Round to 0
PINF	0x80000000	Rounding direction – Round to +Infinity

Table 51. SPARC FPSR Register Bit Settings (Continued)

Value	Bit Setting	Meaning
NINF	0xc0000000	Rounding direction – Round to -Infinity

Using the SPARC FPSR Register

The SPARC processor does not catch floating-point errors by default. You can change the value of the FPSR within TotalView to customize the exception handling for your program.

For example, if your program inadvertently divides by zero, you can edit the bit setting of the FPSR register in the stack frame pane. In this case, you would change the bit setting for the FPSR to include 0x01000000 (as shown in Table 51) so that TotalView traps the "divide by zero" bit. The string displayed next to the FPSR register should now include TEM=(DZ). Now, when your program divides by zero, it receives a SIGFPE signal, which you can catch with TotalView. See Chapter 3, "Setting Up a Debugging Session," on page 35 and "Handling Signals" on page 48 for more information. If you did not set the bit for trapping divide by zero, the processor would ignore the error and set the AEXC=(DZ) bit.

SPARC Floating-Point Format

The SPARC processor supports the IEEE floating-point format.

Alpha

Alpha General Registers

TotalView displays the Alpha general registers in the stack frame pane of the process window. Table 52 describes how TotalView treats each general register, and the actions you can take with each register.

Table 52. Alpha General Purpose Integer Registers

Register	Description	Data Type	Edit	Dive	Specify in Expression
V0	Function value register	**<long>**	yes	yes	**$v0**
T0 – T7	Conventional scratch registers	**<long>**	yes	yes	**$t0 – $t7**
S0 – S5	Conventional saved registers	**<long>**	yes	yes	**$s0 – $s5**
S6	Stack frame base register	**<long>**	yes	yes	**$s6**
A0 – A5	Argument registers	**<long>**	yes	yes	**$a0 – $a5**
T8 – T11	Conventional scratch registers	**<long>**	yes	yes	**$t8 – $t11**
RA	Return Address register	**<long>**	yes	yes	**$ra**
T12	Procedure value register	**<long>**	yes	yes	**$t12**
AT	Volatile scratch register	**<long>**	yes	yes	**$at**
GP	Global pointer register	**<long>**	yes	yes	**$gp**
SP	Stack pointer	**<long>**	yes	yes	**$sp**
ZERO	ReadAsZero/Sink register	**<long>**	no	yes	**$zero**
PC	Program counter	**<code>[]**	no	yes	**$pc**
FP	Frame pointer[1]	**<long>**	no	yes	**$fp**

[1] The Frame Pointer (FP) is a software register that TotalView maintains; it is not an actual hardware register. TotalView computes the value of FP as part of the stack backtrace.

Alpha Floating-Point Registers

TotalView displays the Alpha floating-point registers in the stack frame pane of the process window. Table 53 describes how TotalView treats each floating-point register, and the actions you can take with each register.

Table 53. Alpha Floating-Point Registers

Register	Description	Data Type	Edit	Dive	Specify in Expression
F0 – F1	Floating-point registers (*f* registers), used singly	**<double>**	yes	yes	**$f0 – $f1**
F2 – F9	Conventional saved registers	**<double>**	yes	yes	**$f2 – $f9**
F10 – F15	Conventional scratch registers	**<double>**	yes	yes	**$f10 – $f15**
F16 – F21	Argument registers	**<double>**	yes	yes	**$f16 – $f21**
F22 – F30	Conventional scratch registers	**<double>**	yes	yes	**$f22 – $f30**
F31	ReadAsZero/Sink register	**<double>**	yes	yes	**$f31**
FPCR	Floating-point control register	**<long>**	yes	no	**$fpcr**

Alpha FPCR Register

For your convenience, TotalView interprets the bit settings of the Alpha FPCR register. You can edit the value of the FPCR and set it to any of the bit settings outlined in Table 54.

Table 54. Alpha FPCR Register Bit Settings

Value	Bit Setting	Meaning
SUM	0x8000000000000000	Summary bit
DYN=CHOP	0x0000000000000000	Rounding mode — Chopped rounding mode
DYN=MINF	0x0400000000000000	Rounding mode — Minus infinity
DYN=NORM	0x0800000000000000	Rounding mode — Normal rounding
DYN=PINF	0x0c00000000000000	Rounding mode — Plus infinity
IOV	0x0200000000000000	Integer overflow

Table 54. Alpha FPCR Register Bit Settings (Continued)

Value	Bit Setting	Meaning
INE	0x0100000000000000	Inexact result
UNF	0x0080000000000000	Underflow
OVF	0x0040000000000000	Overflow
DZE	0x0020000000000000	Division by zero
INV	0x0010000000000000	Invalid operation

Alpha Floating-Point Format

The Alpha processor supports the IEEE floating point format.

MIPS

MIPS General Registers

TotalView displays the MIPS general purpose registers in the stack frame pane of the process window. Table 55 describes how TotalView treats each general register, and the actions you can take with each register.

Table 55. MIPS General (Integer) Registers

Register	Description	Data Type[1]	Edit	Dive	Specify in Expression
ZERO	Always has the value 0	**<long>**	no	no	**$zero**
AT	Reserved for the assembler	**<long>**	yes	yes	**$at**
V0 – V1	Function value registers	**<long>**	yes	yes	**$v0 – $v1**
A0 – A7	Argument registers	**<long>**	yes	yes	**$a0 – $a7**
T0 – T3	Temporary registers	**<long>**	yes	yes	**$t0 – $t3**
S0 – S7	Saved registers	**<long>**	yes	yes	**$s0 – $s7**
T8 – T9	Temporary registers	**<long>**	yes	yes	**$t8 – $t9**
K0 – K1	Reserved for the operating system	**<long>**	yes	yes	**$k1 – $k2**
GP	Global pointer	**<long>**	yes	yes	**$gp**
SP	Stack pointer	**<long>**	yes	yes	**$sp**
S8	Hardware frame pointer	**<long>**	yes	yes	**$s8**
RA	Return address register	**<code>[]**	no	yes	**$ra**
MDLO	Multiply/Divide special register, holds least-significant bits of multiply, quotient of divide	**<long>**	yes	yes	**$mdlo**

Table 55. MIPS General (Integer) Registers (Continued)

Register	Description	Data Type[1]	Edit	Dive	Specify in Expression
MDHI	Multiply/Divide special register, holds most-significant bits of multiply, remainder of divide	**<long>**	yes	yes	**$mdhi**
CAUSE	Cause register	**<long>**	yes	yes	**$cause**
EPC	Program counter	**<code>[]**	no	yes	**$epc**
SR	Status register	**<long>**	no	no	**$sr**
VFP	Virtual frame pointer[2]	**<long>**	no	no	**$vfp**

[1] On MIPS, programs compiled either **–64** or **–n32** have 64 bit registers. TotalView uses <long> for –**64** compiled programs and <long long> for **–n32** compiled programs.

[2] The virtual frame pointer is a software register that TotalView maintains. It is not an actual hardware register. TotalView computes the VFP as part of stack backtrace.

MIPS SR Register

For your convenience, TotalView interprets the bit settings of the SR register as outlined in Table 56.

Table 56. MIPS SR Register Bit Settings

Value	Bit Setting	Meaning
0x00000001	IE	Interrupt enable
0x00000002	EXL	Exception level
0x00000004	ERL	Error level
0x00000008	S	Supervisor mode
0x00000010	U	User mode
0x00000018	U	Undefined (implemented as User mode)
0x00000000	K	Kernel mode

Table 56. MIPS SR Register Bit Settings (Continued)

Value	Bit Setting	Meaning
0x00000020	UX	User mode 64-bit addressing
0x00000040	SX	Supervisor mode 64-bit addressing
0x00000080	KX	Kernel mode 64-bit addressing
0x0000FF00	IM=*i*	Interrupt Mask value is *i*
0x00010000	DE	Disable cache parity/ECC
0x00020000	CE	Reserved
0x00040000	CH	Cache hit
0x00080000	NMI	Non-maskable interrupt has occurred
0x00100000	SR	Soft reset or NMI exception
0x00200000	TS	TLB shutdown has occurred
0x00400000	BEV	Bootstrap vectors
0x02000000	RE	Reverse-Endian bit
0x04000000	FR	Additional floating-point registers enabled
0x08000000	RP	Reduced power mode
0x10000000	CU0	Coprocessor 0 usable
0x20000000	CU1	Coprocessor 1 usable
0x40000000	CU2	Coprocessor 2 usable
0x80000000	XX	MIPS IV instructions usable

MIPS Floating-Point Registers

TotalView displays the MIPS floating-point registers in the stack frame pane of the process window. Table 57 describes how TotalView treats each floating-point register, and the actions you can take with each register.

Table 57. MIPS Floating-Point Registers

Register	Description	Data Type	Edit	Dive	Specify in Expression
F0, F2	Hold results of floating-point type function. $f0 has the real part, $f2 has the imaginary part	**<double>**	yes	yes	**$f0**, **$f2**
F1 – F3, F4 – F11	Temporary registers	**<double>**	yes	yes	**$f**1 **– $f3,** **$f4 – $f11**
F12 – F19	Pass single or double precision actual arguments	**<double>**	yes	yes	**$f12 – $f19**
F20 – F23	Temporary registers	**<double>**	yes	yes	**$f20 – $f23**
F24 – F31	Saved registers	**<double>**	yes	yes	**$f24 – $f31**
FCSR	FPU control and status register	**<int>**	yes	no	**$fcsr**

MIPS FCSR Register

For your convenience, TotalView interprets the bit settings of the MIPS FCSR register. You can edit the value of the FCSR and set it to any of the bit settings outlined in Table 58.

Table 58. MIPS FCSR Register Bit Settings

Value	Bit Setting	Meaning
RM=RN	0x00000000	Round to nearest
RM=RZ	0x00000001	Round toward zero
RM=RP	0x00000002	Round toward plus infinity
RM=RM	0x00000003	Round toward minus infinity
flags=(I)	0x00000004	Flag=inexact result

Table 58. MIPS FCSR Register Bit Settings (Continued)

Value	Bit Setting	Meaning
flags=(U)	0x00000008	Flag=underflow
flags=(O)	0x00000010	Flag=overflow
flags=(Z)	0x00000020	Flag=divide by zero
flags=(V)	0x00000040	Flag=invalid operation
enables=(I)	0x00000080	Enables=inexact result
enables=(U)	0x00000100	Enables=underflow
enables=(O)	0x00000200	Enables=overflow
enables=(Z)	0x00000400	Enables=divide by zero
enables=(V)	0x00000800	Enables=invalid operation
cause=(I)	0x00001000	Cause=inexact result
cause=(U)	0x00002000	Cause=underflow
cause=(O)	0x00004000	Cause=overflow
cause=(Z)	0x00008000	Cause=divide by zero
cause=(V)	0x00010000	Cause=invalid operation
cause=(E)	0x00020000	Cause=inexact result
FCC=(0/c)	0x00800000	FCC=Floating-Point Condition Code 0; c=Condition bit
FS	0x01000000	Flush to zero
FCC=(1)	0x02000000	FCC=Floating-Point Condition Code 1
FCC=(2)	0x04000000	FCC=Floating-Point Condition Code 2
FCC=(3)	0x08000000	FCC=Floating-Point Condition Code 3
FCC=(4)	0x10000000	FCC=Floating-Point Condition Code 4

Table 58. MIPS FCSR Register Bit Settings (Continued)

Value	Bit Setting	Meaning
FCC=(5)	0x20000000	FCC=Floating-Point Condition Code 5
FCC=(6)	0x40000000	FCC=Floating-Point Condition Code 6
FCC=(7)	0x80000000	FCC=Floating-Point Condition Code 7

Using the MIPS FCSR Register

You can change the value of the MIPS FCSR register within TotalView to customize the exception handling for your program.

For example, if your program inadvertently divides by zero, you can edit the bit setting of the FCSR register in the stack frame pane. In this case, you would change the bit setting for the FCSR to include 0x400 (as shown in Table 58). The string displayed next to the FCSR register should now include "**enables=(Z)**". Now, when your program divides by zero, it receives a SIGFPE signal, which you can catch with TotalView. See Chapter 3, "Setting Up a Debugging Session," on page 35 and "Handling Signals" on page 48 for more information.

MIPS Floating-Point Format

The MIPS processor supports the IEEE floating point format.

MIPS Delay Slot Instructions

On the MIPS architecture, jump and branch instructions have a "delay slot". This means that the instruction after the jump or branch instruction is executed before the jump or branch is executed.

In addition, there is a group of "branch likely" conditional branch instructions in which the instruction in the delay slot is executed only if the branch is taken.

The MIPS processors execute the jump or branch instruction and the delay slot instruction as an indivisible unit. If an exception occurs as a result of executing the delay slot instruction, the branch or jump instruction is not executed, and the exception appears to have been caused by the jump or branch instruction.

This behavior of the MIPS processors affects both the TotalView instruction step command and TotalView breakpoints.

The TotalView instruction step command will step both the jump or branch instruction and the delay slot instruction as if they were a single instruction.

If a breakpoint is placed on a delay slot instruction, execution will stop at the jump or branch preceding the delay slot instruction, and TotalView will not know that it is at a breakpoint. At this point, attempting to continue the thread which hit the breakpoint without first removing the breakpoint will cause the thread to hit the breakpoint again without executing any instructions. Before continuing the thread, you must remove the breakpoint. If you need to reestablish the breakpoint, you might then use the instruction step command to execute just the delay slot instruction and the branch.

A breakpoint placed on a delay slot instruction of a “branch likely” instruction will be hit only if the branch is going to be taken.

Intel-x86

Intel-x86 General Registers

TotalView displays the Intel-x86 general registers in the stack frame pane of the process window. Table 59 describes how TotalView treats each general register, and the actions you can take with each register.

Table 59. Intel-x86 General Registers

Register	Description	Data Type	Edit	Dive	Specify in Expression
EAX	General registers	**<void>**	yes	yes	**$eax**
ECX		**<void>**	yes	yes	**$ecx**
EDX		**<void>**	yes	yes	**$edx**
EBX		**<void>**	yes	yes	**$ebx**
EBP		**<void>**	yes	yes	**$ebp**
ESP		**<void>**	yes	yes	**$esp**
ESI		**<void>**	yes	yes	**$esi**
EDI		**<void>**	yes	yes	**$edi**
CS	Selector registers	**<void>**	no	no	**$cs**
SS		**<void>**	no	no	**$ss**
DS		**<void>**	no	no	**$ds**
ES		**<void>**	no	no	**$es**
FS		**<void>**	no	no	**$fs**
GS		**<void>**	no	no	**$gs**
EFLAGS		**<void>**	no	no	**$eflags**

Table 59. Intel-x86 General Registers (Continued)

Register	Description	Data Type	Edit	Dive	Specify in Expression
EIP	Instruction pointer	**<code>[]**	no	yes	**$eip**
FAULT		**<void>**	no	no	**$fault**
TEMP		**<void>**	no	no	**$temp**
INUM		**<void>**	no	no	**$inum**
ECODE		**<void>**	no	no	**$ecode**

Intel-x86 Floating-Point Registers

TotalView displays the x86 floating-point registers in the stack frame pane of the process window. Table 60 describes how TotalView treats each floating-point register, and the actions you can take with each register.

Table 60. Intel-x86 Floating-Point Registers

Register	Description	Data Type	Edit	Dive	Specify in Expression
ST0	ST(0)	**<extended>**	yes	yes	**$st0**
ST1	ST(1)	**<extended>**	yes	yes	**$st1**
ST2	ST(2)	**<extended>**	yes	yes	**$st2**
ST3	ST(3)	**<extended>**	yes	yes	**$st3**
ST4	ST(4)	**<extended>**	yes	yes	**$st4**
ST5	ST(5)	**<extended>**	yes	yes	**$st5**
ST6	ST(6)	**<extended>**	yes	yes	**$st6**
ST7	ST(7)	**<extended>**	yes	yes	**$st7**
FPCR	Floating-point control register	**<void>**	yes	no	**$fpcr**

Table 60. Intel-x86 Floating-Point Registers (Continued)

Register	Description	Data Type	Edit	Dive	Specify in Expression
FPSR	Floating-point status register	**<void>**	no	no	**$fpsr**
FPTAG	Tag word	**<void>**	no	no	**$fptag**
FPIOFF	Instruction offset	**<void>**	no	no	**$fpioff**
FPISEL	Instruction selector	**<void>**	no	no	**$fpisel**
FPDOFF	Data offset	**<void>**	no	no	**$fpdoff**
FPDSEL	Data selector	**<void>**	no	no	**$fpdsel**

Intel-x86 FPCR Register

For your convenience, TotalView interprets the bit settings of the FPCR and FPSR registers.

You can edit the value of the FPCR and set it to any of the bit settings outlined in Table 61.

Table 61. Intel-x86 FPCR Register Bit Settings

Value	Bit Setting	Meaning
RC=NEAR	0x0000	To nearest rounding mode
RC=NINF	0x0400	Toward negative infinity rounding mode
RC=PINF	0x0800	Toward positive infinity rounding mode
RC=ZERO	0x0c00	Toward zero rounding mode
PC=SGL	0x0000	Single precision rounding
PC=DBL	0x0080	Double precision rounding
PC=EXT	0x00c0	Extended precision rounding
EM=PM	0x0020	Precision exception enable

Table 61. Intel-x86 FPCR Register Bit Settings (Continued)

Value	**Bit Setting**	**Meaning**
EM=UM	0x0010	Underflow exception enable
EM=OM	0x0008	Overflow exception enable
EM=ZM	0x0004	Zero divide exception enable
EM=DM	0x0002	Denormalized operand exception enable
EM=IM	0x0001	Invalid operation exception enable

Using the Intel-x86 FPCR Register

You can change the value of the FPCR within TotalView to customize the exception handling for your program.

For example, if your program inadvertently divides by zero, you can edit the bit setting of the FPCR register in the stack frame pane. In this case, you would change the bit setting for the FPCR to include 0x0004 (as shown in Table 61) so that TotalView traps the “divide by zero” bit. The string displayed next to the FPCR register should now include EM=(ZM). Now, when your program divides by zero, it receives a SIGFPE signal, which you can catch with TotalView. See Chapter 3 of the *TotalView User’s Guide* for information on handling signals. If you did not set the bit for trapping divide by zero, the processor would ignore the error and set the EF=(ZE) bit in the FPSR.

Intel-x86 FPSR Register

The bit settings of the Intel-x86 FPSR register are outlined in Table 62.

Table 62. Intel-x86 FPSR Register Bit Settings

Value	**Bit Setting**	**Meaning**
TOP=<*i*>	0x3800	Register <*i*> is top of FPU stack
B	0x8000	FPU busy
C0	0x0100	Condition bit 0

Table 62. Intel-x86 FPSR Register Bit Settings (Continued)

Value	Bit Setting	Meaning
C1	0x0200	Condition bit 1
C2	0x0400	Condition bit 2
C3	0x4000	Condition bit 3
ES	0x0080	Exception summary status
SF	0x0040	Stack fault
EF=PE	0x0020	Precision exception
EF=UE	0x0010	Underflow exception
EF=OE	0x0008	Overflow exception
EF=ZE	0x0004	Zero divide exception
EF=DE	0x0002	Denormalized operand exception
EF=IE	0x0001	Invalid operation exception

Intel-x86 Floating-Point Format

The Intel-x86 processor supports the IEEE floating point format.

Glossary

action point A point in a program where a breakpoint, evaluation point, or event point has been set during a TotalView session.

address space A region of memory that contains code and data from a program. One or more threads can run in an address space. A process normally contains an address space.

automatic process acquisition TotalView automatically detects the many processes that parallel and distributed programs run in and attaches to them automatically so you don't have to attach to them manually. This process is called *automatic process acquisition*. If the process is on a remote machine, automatic process acquisition also automatically starts the TotalView debugger server.

breakpoint A point in a program where execution can be conditionally suspended to permit examination and manipulation of data.

child process A process created by another process (see parent process) when that other process calls fork().

cluster debugging The action of debugging a program that is running on a cluster of hosts in a network. Typically, the hosts are homogeneous.

core file A file containing the contents of memory and a list of thread registers. The operating system dumps (creates) a core file whenever a program exits because of a severe error (such as an attempt to store into an invalid address).

cross debugging A special case of remote debugging where the host platform and the target platform are different types of machines.

data-set A set of array elements generated by TotalView and sent to the Visualizer.

dbelog library A library of routines for creating event points and generating event logs from within TotalView. To use event points, you must link your program with both the **dbelog** and **elog** libraries.

dbfork library A library of special versions of the **fork**() and **execve**() calls used by the TotalView debugger to debug multiprocess programs. Programs that call one of the **fork**(), **vfork()**, or **execve**() routines must be linked with the dbfork library.

debugger server *See* the glossary entry for **tvdsvr process**.

distributed debugging The action of debugging a program that is running on more than one host in a network. The hosts can be homogeneous or heterogeneous. For example, programs written with message passing libraries such as Parallel Virtual Machine (PVM) or Parallel Macros (PARMACS) run on more than one host.

dive stack A series of nested dives that were performed in the same variable window. The number of right angle brackets (>) in the upper left hand corner of a variable window indicates the number of nested dives on the dive stack. Each time that you undive, TotalView pops a dive from the dive stack and decrements the number of right angle brackets shown in the variable window.

diving The action of displaying more information about an item. For example, if you dive into a variable in TotalView, a window appears with more information about the variable.

editing cursor A black rectangle that appears when a TotalView field is selected for editing. You use field editor commands to move the editing cursor.

elog library A library of routines for generating event logs from multiprocess programs. The event logs can be displayed and analyzed with the Gist application.To use event points, you must link your program with both the **dbelog** and **elog** libraries.

evaluation point A point in the program where TotalView evaluates a code fragment without stopping the execution of the program.

event log A file containing a record of events for each process in a program.

event point A point in the program where TotalView writes an event to the event log for later analysis using Gist.

extent The number of elements in the dimension of an array. For example, a Fortran array of integer(7,8) has an extent of 7 in one dimension (7 rows) and an extent of 8 in the other dimension (8 columns).

field editor A basic text editor that is part of TotalView's interface. The field editor supports a subset of GNU Emacs commands.

gridget A dotted grid in the tag field that indicates you can set an action point on the instruction.

host machine The machine on which the TotalView debugger is running.

lower bound The first element in the dimension of an array or the slice of an array. By default, the lower bound of an array is 0 in C and 1 in Fortran, but the lower bound can be any number, including negative numbers.

message queue A list of messages sent and received by message passing programs.

MPICH MPI/Chameleon (Message Passing Interface/Chameleon, most commonly referred to as MPICH) is a freely-available and portable MPI implementation. MPICH was written as a collaboration between Argonne National Lab and Mississippi State University. For more information, see **http://www.mcs.anl.gov/mpi**.

mutex Mutual exclusion. A collection of techniques for sharing resources so that different uses do not conflict and cause unwanted interactions.

native debugging The action of debugging a program that is running on the same machine as TotalView.

nested dive window A TotalView window that results from diving into an item in a variable window. A nested dive window replaces the contents of the variable window and has an undive symbol in its title bar. Diving on the undive symbol returns the original contents of the variable window.

parcel The number of bytes required to hold the shortest instruction for the target architecture.

parent process A process that calls fork() to spawn other processes (usually called child processes).

PARMACS library A message passing library for creating distributed programs that was developed by the German National Research Centre for Computer Science.

process Consists of an address space and a list of one or more threads running in that address space.

process group A group of processes associated with a multiprocess program. Includes program groups and share groups.

process window The main TotalView window for a process, which consists of three panes: the stack trace, the stack frame, and the source code for the program.

program group A group of processes that includes the parent process and all related processes. A program group includes children that were forked (processes that share the same source code as the parent) and children that were forked with a subsequent call to execve() (processes that do *not* share the same source code as the parent). Contrast with share group.

PVM library Parallel Virtual Machine library. A message passing library for creating distributed programs that was developed by the Oak Ridge National Laboratory and the University of Tennessee.

remote debugging The action of debugging a program that is running on a different machine than TotalView. The machine on which the program is running can be located many miles away from the machine on which TotalView is running.

root window A TotalView window displaying the process ID, status (e.g., at breakpoint or stopped), name, and current routine executing for each process being debugged.

serial line debugging A form of remote debugging where TotalView and the TotalView Debugger Server communicate over a serial line.

share group A group of processes that includes the parent process and any related processes that share the same source code as the parent. Contrast with program group.

signals Messages informing processes of asynchronous events, such as serious errors. The action the process takes in response to the signal depends on the type of signal and whether or not the program includes a signal handler routine, a routine that traps certain signals and determines appropriate actions to be taken by the program.

single step The action of executing a single statement and stopping (as if at a breakpoint).

slice A subsection of an array, which is expressed in terms of a lower bound, upper bound, and stride. Displaying a slice of an array can be useful when working with very large arrays, which is often the case in Fortran programs.

stack A portion of computer memory and/or registers used to hold information temporarily. The stack consists of a linked list of stack frames that holds return locations for called routines, routine arguments, local variables, and saved registers.

stack frame A section of the stack that contains the local variables, arguments, contents of the registers used by an individual routine, a frame pointer pointing to the previous stack frame, and the value of the Program Counter (PC) at the time the routine was called.

stack trace A sequential list of each currently active routine called by a program and the frame pointer pointing to its stack frame.

stride The interval between array elements in a slice and the order in which the elements are displayed. If the stride is 1, every element between the lower bound and upper bound of the slice is displayed. If the stride is 2, every other element is displayed. If the stride is –1, every element between the upper bound and lower bound (reverse order) is displayed.

symbol table A table of symbolic names (such as variables or functions) used in a program and their memory locations. The symbol table is part of the executable object generated by the compiler (with the **–g** switch) and is used by debuggers to analyze the program.

tag field The left margin in the source code pane of the TotalView process window containing boxed line numbers marking the lines of source code that actually generate executable code.

target machine The machine on which the process to be debugged is running.

thread An execution context that normally contains a set of private registers and a region of memory reserved for an execution stack. A thread runs in an address space.

tvdsvr process The TotalView Debugger Server process, which facilitates remote debugging by running on the same machine as the executable and communicating with TotalView over a TCP/IP port or serial line.

undiving The action of displaying the previous contents of a window, instead of the contents displayed for the current dive. To undive, you dive on the undive icon in the upper right-hand corner of the window.

upper bound The last element in the dimension of an array or the slice of an array.

variable window A TotalView window displaying the name, address, data type, and value of a particular variable.

visualizer process A process that works with TotalView in a separate window allowing you to see a graphical representation of program array data.

Index